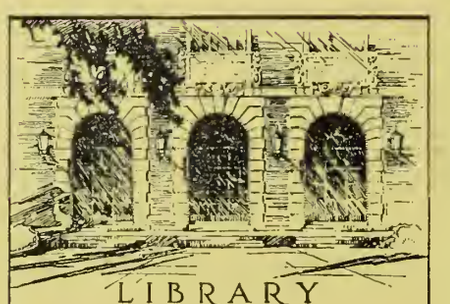


LIBRARY
OF THE
UNIVERSITY
OF ILLINOIS

510.84
I16r
no. 195-203
cop. 2

The person charging this material is responsible for its return on or before the **Latest Date** stamped below.

Theft, mutilation and underlining of books are reasons for disciplinary action and may result in dismissal from the University.

UNIVERSITY OF ILLINOIS LIBRARY AT URBANA-CHAMPAIGN

~~JUL 15 1971~~
AUG 12 1971

AUG · 0 RECD
due 6/27
anytime
JUN 27 RECD

MAR 04 1982
BUILDING USE ONLY
MAR 04 1982

L161—O-1096

510.84
I 6r
no. 195
cop. 2

REPORT NO. 195

COO-1018-1069

195-202

METHODS OF BINARY ADDITION

by

Roger E. Wiegel

February 1, 1966

DEPARTMENT OF COMPUTER SCIENCE · UNIVERSITY OF ILLINOIS · URBANA, ILLINOIS

The person charging this material is responsible for its return on or before the **Latest Date** stamped below.

Theft, mutilation, and underlining of books are reasons for disciplinary action and may result in dismissal from the University.

University of Illinois Library

FEB 26 1963
FEB 6 RECD

L161— O-1096

Report No. 195

METHODS OF BINARY ADDITION*

by

Roger E. Wiegel

February 1, 1966

Department of Computer Science
University of Illinois
Urbana, Illinois 61801

* This work was supported in part by Contract No. US AEC AT(11-1)1018 and was submitted in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering, January, 1966.

510.84
I16
no.195-203
Cop. 2

ACKNOWLEDGMENT

The author wishes to express his sincere thanks to Professor S. R. Ray whose advice and assistance were greatly appreciated in the preparation of this paper.

Thanks are also due to Miss Bonnie Malcor for typing the final draft. The excellent drawings in this report are a result of the good work of Mrs. L. A. Prendergast of the Department of Computer Science Drafting Department.

TABLE OF CONTENTS

Page

1. INTRODUCTION 1

2. HISTORY OF BINARY ADDERS 3

3. RIPPLE CARRY ADDER 5

3.1 Half-Adder 5
3.2 Full-Adder 5
3.3 Ripple Carry Adder 8

4. CARRY LOOKAHEAD ADDER 10

4.1 Principles of Operation 10
4.2 Adder Logic 11
4.3 Second Level Lookahead 14
4.4 Lookahead Variations 15
4.5 Lookahead Adder Summary 18

5. CARRY SKIP ADDER 20

5.1 Basic Skip Circuit 20
5.2 Extension of Carry Skip 22
5.3 Conclusion 26

6. THE CONDITIONAL SUM ADDER 30

6.1 Basic Theory and Logic 30
6.2 Improved Conditional Sum Logic 32
6.3 Further Speed Increases 38
6.4 Conditional Sum Adder Summary 42

7. THE ASYNCHRONOUS ADDER 43

7.1 Principles of Operation 43
7.2 Asynchronous Carry Logic 44
7.3 Self-Timing Adder Stage 46
7.4 Variable-Time Adder Summary 48

8. SEPARATE CARRY-STORAGE ADDER 49

8.1 Principles of Operation 49
8.2 Adder Logic 51
8.3 Summary 54

9. SUMMARY OF ADDER-TYPES 55

9.1 Adder Design Notes 56
9.2 Future 57

APPENDIX 1 59

APPENDIX 2 61

BIBLIOGRAPHY 64

1. INTRODUCTION

An "adder" is defined as the physical realization of some logical scheme to achieve the (binary) addition of two numbers within the framework of some larger system. The purpose of this paper is to present and discuss the logical organization and operating principles of various types of parallel binary adders. Since neither the physical realization of the adder nor the larger system surrounding the adder are our principal concern, certain assumptions will be made about the environment in which the adder is embedded.

First, it is assumed that the system operates, in parallel, on binary number fields, n bits in length, as contrasted to a serial machine, which operates on bits of a binary number one at a time, i.e., in series. Examples of parallel machines are computers such as the ILLIAC II, IBM 7090, and the CDC 6600. A series computer is one such as the EDVAC, which operates on 44-bit words.

The number field bits are represented by X_1, ..., X_n, with X_n being the least significant (right-most) bit. The number representation employed in the system may be any one of signed-magnitude, 1's complement (diminished radix complement), or 2's complement (radix complement). The basic adder mechanism functions independently of number representation. Auxiliary operations such as pre- and post-addition complementing logic, overflow, underflow, and zero detection logic, and end-around and low-order carry injection circuitry are performed in the peripheral circuitry. Flores' book [5]* contains a complete summary and examples of the rules for addition, subtraction, etc. in the various number systems.

---

* Numbers in brackets refer to the corresponding entry in the Bibliography.

Furthermore, it is assumed that both true and complemented inputs are available from the necessary input and output buffering registers which are provided along with control and gating logic exterior to the adder.

For each adder-type discussed, two or more logical design variations will be presented. The first will best illustrate the operating and logical principles of the adder in question, and will be based on the unrealistic assumption that the physical elements have an indefinitely large fanin and fanout capability. The alternate design(s) will be guided by physical constraints, i.e., elements are capable of fanins of 2 to 10; fanouts of 1 for an AND or OR, and fanouts of 2 to 20 for a NOT (or NAND). Direct-coupled rather than pulse-type logic will be used.

2. HISTORY OF BINARY ADDERS

Addition is one of the more common processes occurring in general purpose and scientific computers. It is not uncommon to find two or more adders of different types in a single machine. These adders are often used several times during the execution of the various instructions. Since the adder is so central to the operation of a computer, it pays to carefully choose and design an adder to match the machine and its projected use.

The ENIAC, put into operation in 1945, was the first chiefly electronic computer. It had a word length of 10 decimal digits plus sign and effected addition by using 10 stage decade counters. ENIAC used 10's complement representation of numbers for subtraction. Its add time was about 100 μsec. EDVAC (1949) was a serial machine with a word length of 44 bits.

ILLIAC I, completed in 1952, was based upon the IAS computer described by Burkes, et al [4]. Additions on ILLIAC's 40-bit words were performed in parallel, using a ripple carry adder, with an add time of about 15 μsec. A carry could propagate through eight stages in about 1.1 μsec, and the sum for any given bit was formed within 1/4 μsec after the arrival of the carry. In the worst case, a carry had to ripple the full, 40-position length of the adder before the addition could be completed.

It can be seen (especially in ILLIAC I) that most of the time required by adders is needed for carry propagation. This was one of the more serious obstacles to overcome in speeding up the addition process as computers were made faster.

One obvious way to decrease carry propagation time (increase speed) is to use faster components -- i.e., faster transistors, diodes, etc. -- but there are limits in doing this, since as the speed of a device increases,

so does its cost. So the designer has two paths open to him: he may develop faster components and circuitry, or he may organize slower components into faster, more efficient systems. It is the latter course which will be followed in this paper, which describes several methods which have been developed for reducing the carry propagation time.

3. RIPPLE CARRY ADDER

The ripple carry adder is fundamental to all adder-types. It was one of the first adders used in computers. From it, all other adder-types have evolved.

Before discussing the ripple carry adder, its building blocks, the half- and full-adders, will be introduced.

3.1 Half-Adder

The basic building block of the ripple carry adder is the half-adder. When two binary digits X_i and Y_i are added, a sum digit, S_i, and an output carry digit, C_{i-1}, are obtained. The relations between X_i, Y_i, S_i, and C_{i-1} are given in the truth table below. The logical realization with "ideal" elements* is shown in Figure 1. The block diagram which represents the half-adder is also given.

Half-Adder Truth Table

X_i	Y_i	C_{i-1}	S_i
0	0	0	0
0	1	0	1
0	0	0	1
0	1	1	1

$$S_i = (X_i \vee Y_i)\ \overline{X_i\ Y_i} = X_i\ \bar{Y}_i \vee \bar{X}_i\ Y_i$$

$$C_i = X_i\ Y_i$$

3.2 Full-Adder

To form the sum of two binary digits when an input carry, C_i, is present, two half-adders, plus an additional OR to assemble the output carry, C_{i-1}, are necessary. The truth table from which the full-adder logic

---

* Definitions of all logic symbols are given in Appendix 1.

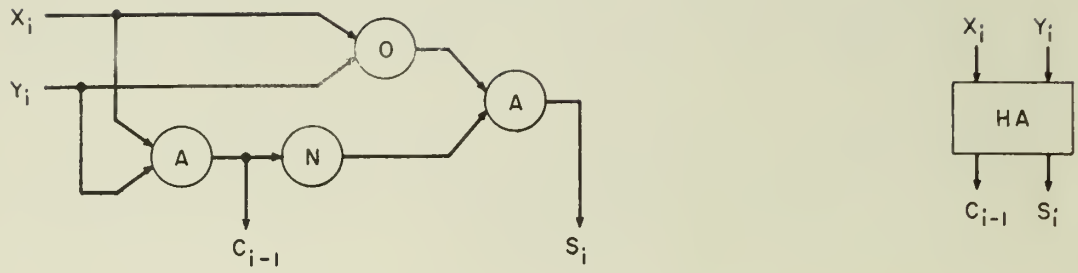


FIGURE 1. HALF-ADDER

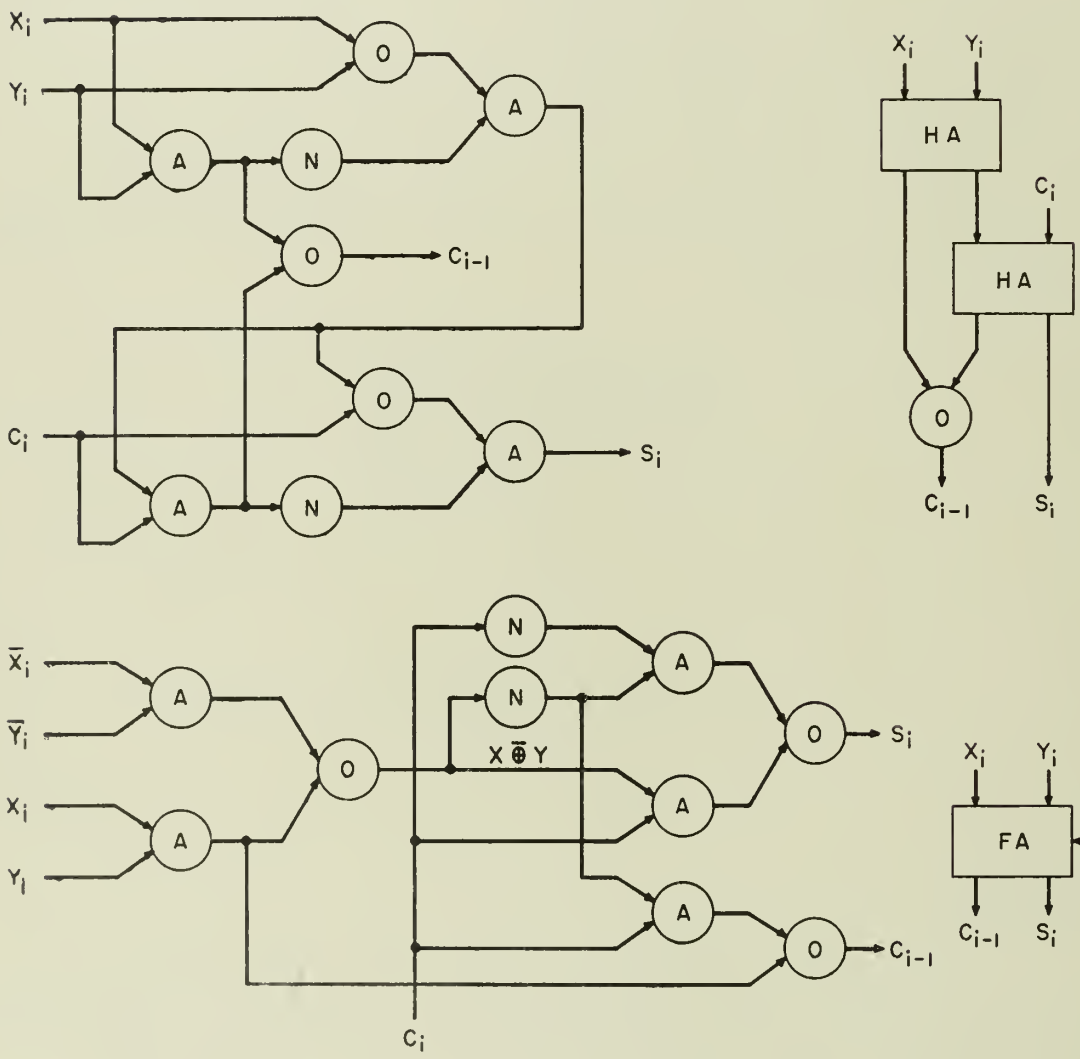


FIGURE 2. ALTERNATE LOGICS FOR ONE BIT OF FULL BINARY ADDER

is realized is:

Full-Adder Truth Table

X_i	Y_i	C_i	C_{i-1}	S_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Figure 2 illustrates the full-adder logic. The partial sum output from the first half-adder and the carry input to the stage form the inputs to the second half-adder. An output carry (C_{i-1}) may come from either half-adder and arises whenever a majority of the inputs (X_i, Y_i, and C_i) to the stage are '1'. The sum, S_i, is '1' when only one or all three of the inputs are '1'.

Equations for the full-adder are:

$$S_i = X_i \, \overline{Y}_i \, \overline{C}_i \vee \overline{X}_i \, Y_i \, \overline{C}_i \vee \overline{X}_i \, \overline{Y}_i \, C_i \vee X_i \, Y_i \, C_i$$

or

$$S_i = (X_i \oplus Y_i) \cdot \overline{C}_i \vee (X_i \overline{\oplus} Y_i) \cdot C_i$$

and

$$C_{i-1} = X_i \, Y_i \vee X_i \, C_i \vee Y_i \, C_i$$

or

$$C_{i-1} = (X_i \oplus Y_i) \cdot C_i \vee X_i \, Y_i$$

Figure 3 gives an alternate logical realization, using NANDS, for one bit position of the full-adder. This logic is better than that of Figure 2 in that the NANDS provide signal level restoration and amplification, for the carry in particular. Collector logic is used in two instances to produce a negative OR function, i.e., if the output of either NAND of the common pair falls to '0', the output from the common junction goes to '0'.

3.3 Ripple Carry Adder

The one bit full-adder blocks of Figure 2 or 3 can be interconnected as shown in Figure 4 to form an n-bit ripple carry adder. The term "ripple carry" is used since the sum bit at stage i cannnot be formed until the carry C_i is input. C_i depends on the inputs to stage i+1, which depend on stage i+2, etc. Hence, carries must ripple through from the low-order stage (n) to the high-order stage (bit 1) in order to complete an addition.

If the NAND logic of Figure 3 is used, two delays per stage are needed to propagate the carry (one delay is the time needed for a signal to pass through a NAND circuit), for a total of 2n delays* to form the sum in an n-bit adder.

The speed of addition in the ripple carry adder is limited by the maximum time required to propagate a carry through the adder. In modern computers, this time can be excessive. The obvious way to speed up the addition process is to decrease the carry propagation time. The following sections describe different methods of reducing carry propagation time.

---

* 2n = 2 delays in the first stage to generate the carry, plus two in the middle stages to propagate it, plus two in the last stage to form the sum.

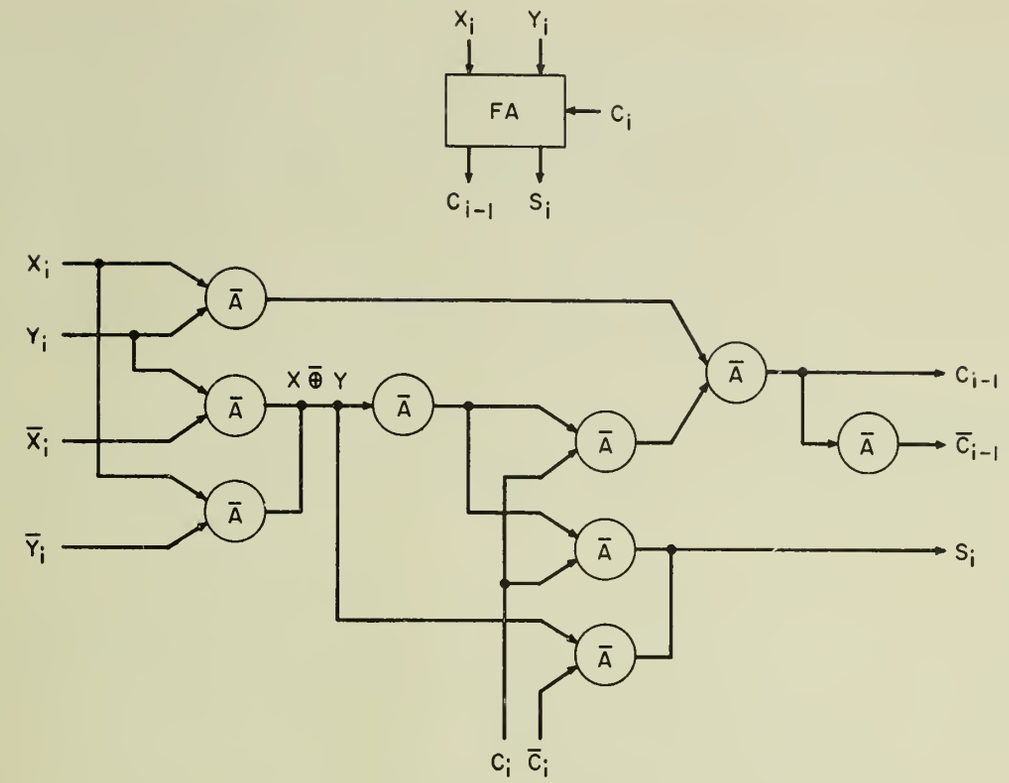


FIGURE 3. ONE POSITION OF FULL BINARY ADDER USING NAND LOGIC

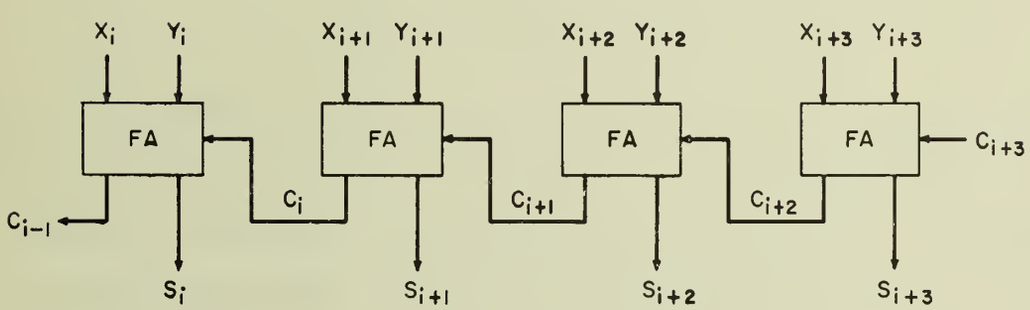


FIGURE 4. 4-BIT SECTION OF RIPPLE CARRY ADDER

4. CARRY LOOKAHEAD ADDER

The carry lookahead adder (or simultaneous carry or distant carry adder) is a direct descendant of the ripple carry adder. Instead of waiting for carries to ripple through to some stage i, logic is employed which "looks ahead" to see what the true input carry to stage i will be without waiting for it to ripple up to the stage.

4.1 Principles of Operation

In the ripple carry adder, the carry into stage i was defined as:

$$C_i = X_{i-1}\, Y_{i-1} \vee C_{i-1}\, (X_{i-1} \vee Y_{i-1})$$

As mentioned above, the formation of each carry digit must await the generation of the next lower-order carry.

If the following functions are defined:

$$G_i = X_i\, Y_i$$

$$T_i = X_i \vee Y_i$$

then the carry expression becomes

$$C_i = G_{i-1} \vee T_{i-1}\, C_{i-1}$$

Note that $X_i \oplus Y_i$ will work as well as $X_i \vee Y_i$ in the expression for T_i, since if both X_i and Y_i are '1', then G_i is '1' and it does not matter if T_i is '1' or '0'. The carry expression now consists of two parts: G, which is produced independently of an input carry, and T, which causes a carry to be generated on the condition that there is an input carry. In effect, G is a generated carry, and T is a propagated (transmitted) carry.

From this description, the carry into any stage i can be written in terms of the G's and T's of the preceding stages so that any bit position can "look ahead" to see what its true carry input will be. The lookahead equation is:

$$C_i = G_{i-1} \vee T_{i-1} C_{i-1}$$

$$= G_{i-1} \vee T_{i-1} G_{i-2} \vee T_{i-1} T_{i-2} C_{i-2}$$

$$= G_{i-1} \vee T_{i-1} G_{i-2} \vee T_{i-1} T_{i-2} G_{i-3} \vee T_{i-1} T_{i-2} T_{i-3} C_{i-3}$$

This expression can be continued as far as desired, limited only by physical considerations.

4.2 Adder Logic

Figure 5b illustrates the application of the carry lookahead principle to an adder section of 4 bits. (Figure 5a is the modified full-adder logic used in the FA' blocks of Figure 5b.) X_i and Y_i are the addend and augend inputs, S_i is the sum output from bit position i, C_{in} is the carry input from the preceding lookahead group, and C_{out} is the output carry to the next group.

If the logic of Figures 5a, 5b is replaced by the equivalent NAND logic, an n-bit adder can be assembled using n/4 lookahead groups. This adder would have full carry lookahead within 4-bit groups, with ripple carry between groups. The adder would need two delays to generate all T's and G's; two delays per group are then needed to propagate the carries through the adder. Two of the four delays needed for sum generation are taken simultaneously with the output carry generation from the last group. Thus, for a 40-bit adder (ten groups), 24 delays are required to complete an addition. In this case, a 39 per cent increase in hardware

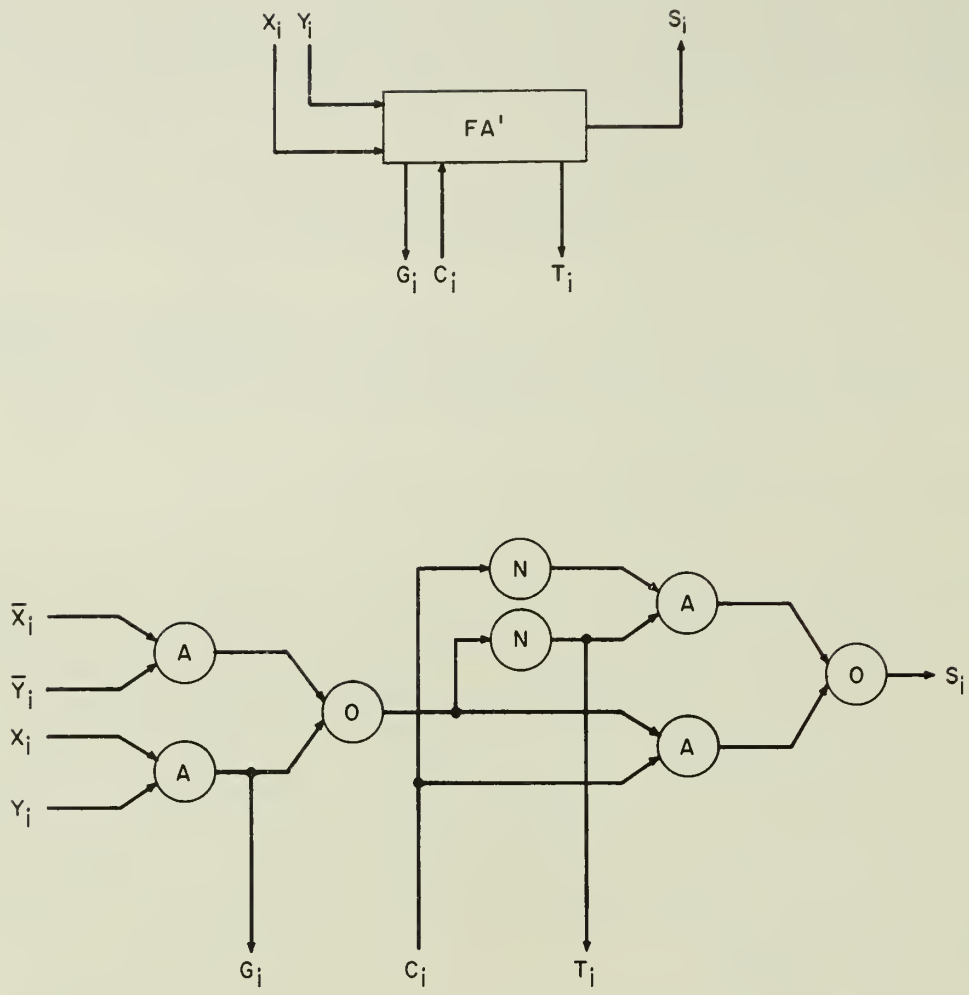


FIGURE 5a. MODIFIED FULL-ADDER OF FIGURE 2 FOR USE IN CARRY LOOKAHEAD ADDER

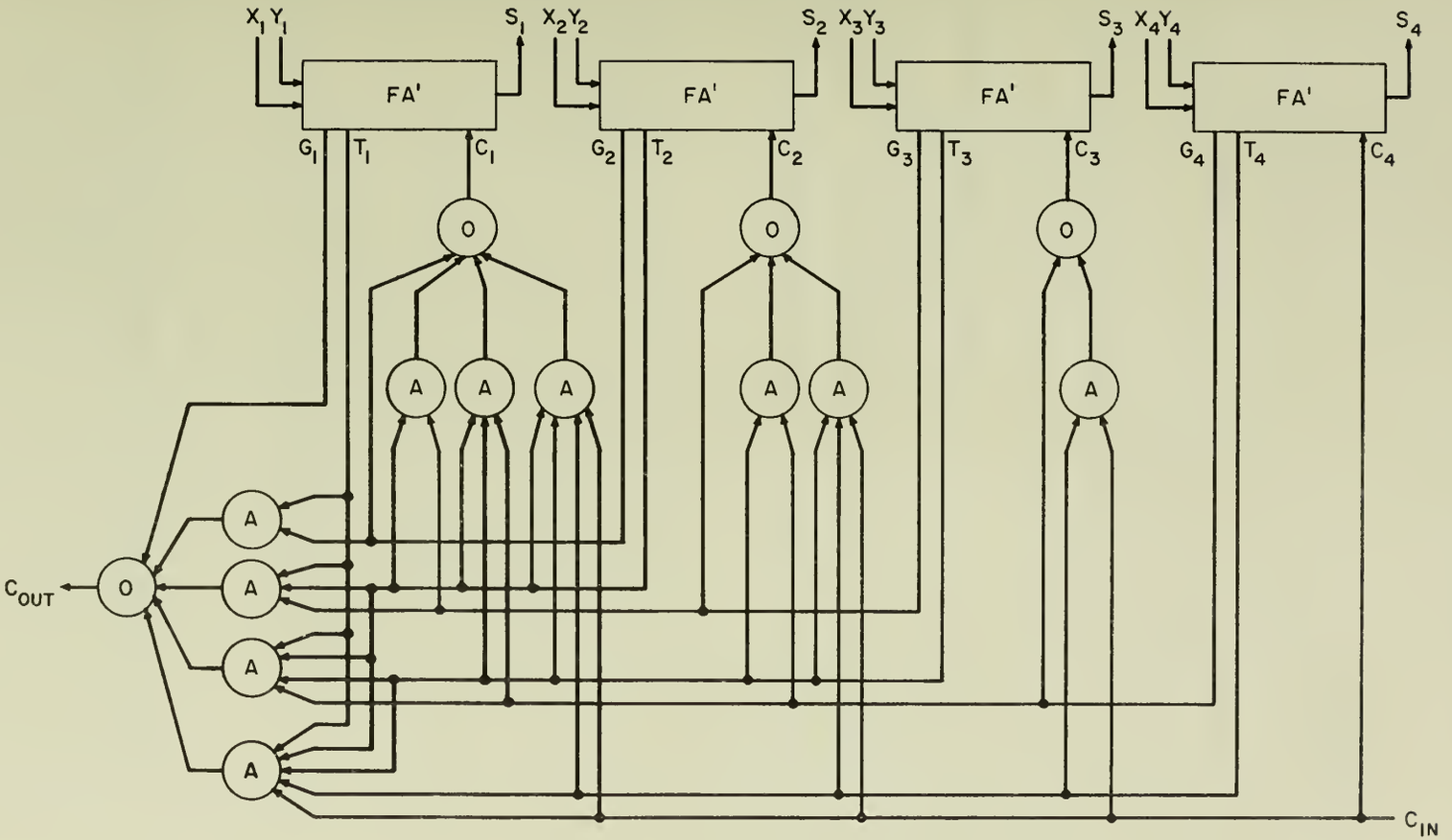


FIGURE 5b. 4-BIT ADDER SECTION
WITH FULL CARRY LOOKAHEAD

over the ripple carry adder will increase the addition speed by a factor of 3.3.*

4.3 Second Level Lookahead

Carry lookahead between bits in a group is called level one lookahead; between groups within a section (two or more groups) level two lookahead, and between sections, level three.

The philosophy behind level one lookahead is that since it is easy to tell whether or not a given bit position will generate ($X_i\ Y_i$ = '1') or propagate ($X_i \vee Y_i$ = '1') a carry, the carry into a bit position can be determined from the G's and T's of lower-ordered positions. This idea can be extended from single bits to groups of bits; so it can be determined whether or not a given group will generate a carry, propagate through, or absorb an incident carry.

A section can be defined as consisting of three or four groups; the 4-bit groups of Figure 5b in the present case. The equations for the level two output carry, C_0, from the section are, for a three group section:

$$C_0 = G_1 \vee T_1 G_2 \vee T_1 T_2 G_3 \vee T_1 T_2 T_3 G_4 \vee P_1 G_5 \vee P_1 T_5 G_6 \vee$$
$$P_1 T_5 T_6 G_7 \vee P_1 T_5 T_6 T_7 G_8 \vee P_1 P_2 G_9 \vee P_1 P_2 G_9 \vee$$
$$P_1 P_2 G_9 \vee P_1 P_2 T_9 G_{10} \vee P_1 P_2 T_9 T_{10} G_{11} \vee P_1 P_2 T_9 T_{10} T_{11} G_{12} \vee$$
$$P_1 P_2 P_3 C_{in}$$

where the T's and G's are defined above, and the P's are the propagate functions

---

* Appendix 2 contains a chart which compares various adder types with the ripple carry adder. Hardware and speed comparisons are given there.

$$P_1 = T_1 T_2 T_3 T_4$$

$$P_2 = T_5 T_6 T_7 T_8$$

and $$P_3 = T_9 T_{10} T_{11} T_{12}$$

If a carry is generated anywhere within the 12 bits, and all intervening bits would transmit it, the carry will immediately appear as C_0. All of the above functions, except the P_i's, are available in the level one logic, and the P's can be added at little cost.

The above discussion can be extended to level two lookahead over any number of groups in a section, or to level three lookahead between sections. The realization of the above equations in hardware is similar to the logic of Figure 5c and will not be given. Level three lookahead, in general, will be of little use or benefit in adders operating on word lengths of fewer than 40 to 50 bits.

4.4 Lookahead Variations

Figure 6 illustrates three of the many possible ways of segmenting a 40-bit adder into sections. The first contains sections of 1, 3, 3, and 3 groups, the second contains 2, 4, and 4 groups, and the third has 1, 2, 3, and 4 groups per section. Each group is comprised of 4 bits with full lookahead as shown in Figure 5b.

Carries ripple through the second level lookahead units, and, from them, through the individual groups in each section. The maximum length carry ripple path in Figure 6a traverses second level functions S2 and S3, then through all groups fed by the S2 function (G4, G3, and G2).

In all variants, two delays are needed to form all G and T functions and two delays per section lookahead unit (S1, S2, S3) are necessary for carry propagation. Each of the left-most groups (G1 and G2) requires two delays for carry propagation and finally, sum generation requires two more delays.

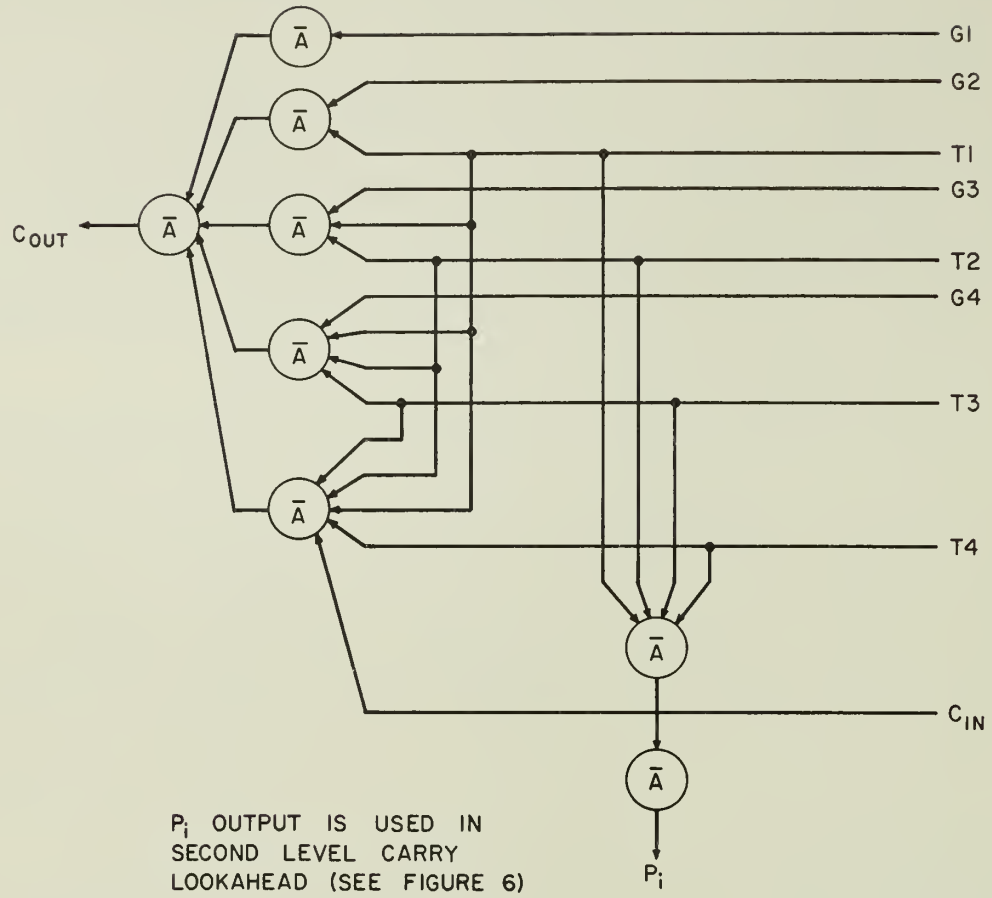


FIGURE 5 c. ALTERNATE CARRY GENERATION LOGIC FOR CARRY LOOKAHEAD ADDER SECTION

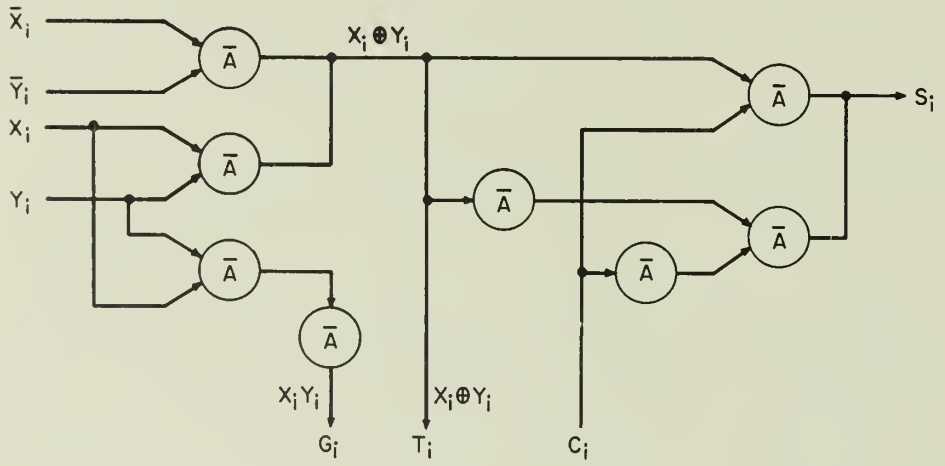


FIGURE 5d. ADDER LOGIC OF FIGURE 3 MODIFIED FOR USE IN CARRY LOOKAHEAD ADDER

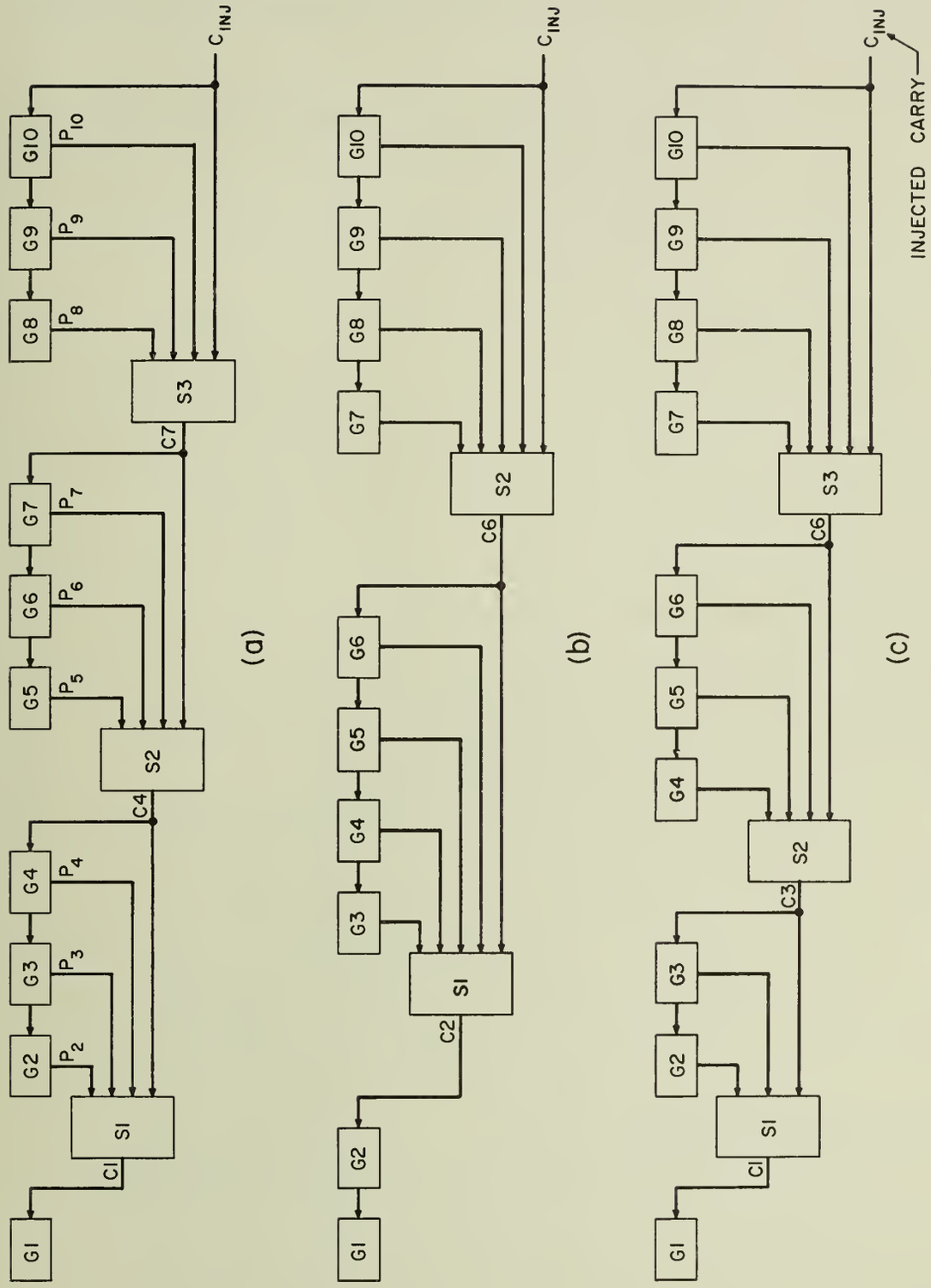


FIGURE 6. ALTERNATE ARRANGEMENTS FOR GROUPS WITHIN SECTIONS OF LEVEL TWO CARRY LOOKAHEAD ADDER

For version (a) of Figure 6, 14 delays total are needed; version (b) of the 40-bit adder likewise requires 14 delays; version (c), however, can complete an addition in 12 delays although it needs no more hardware than (a) or (b). It is seen that a rearrangement of groups can decrease the add time.

For the lookahead adder using level two lookahead, a 46 per cent hardware increase cuts the add time by a factor of 7 over the ripple carry adder. Further time savings may be realized if the group size were increased from 4 to 5 or 8 bits (for the 40-bit adder).

The lookahead adders described above can be modified in various ways to suit the needs and desires of the logic designer. One possible variant might have carry lookahead within groups with ripple carry between groups. Some of the other changes which could be made if a larger or smaller size adder, or faster or slower operation is desired, are (1) use ripple carry within groups and full lookahead between groups; (2) use full lookahead within groups and between groups; and (3) use the logic of the completion recognition adder (Section 7) to detect the completion of the carry propagation between groups.

The speeds and hardware requirements for two variations of this adder are given in Appendix 2.

Most of the design ideas and methods covered in the following section on carry skip adders applies equally well to the design of carry lookahead adders, so nothing more will be said here about adder design.

4.5 Lookahead Adder Summary

The carry lookahead adder, first described by Weinberger and Smith [18] in 1956, is basically a fixed-time adder, i.e., the addition is completed in a fixed allotted time. If modification (3), above, is employed, the adder becomes a variable-time device.

MacSorley [10] gives a comparison of five lookahead adder variants using 50 and 100 bits.

The lookahead adder has an ultimate speed capability of 8 delays. The amount of logic required would be quite large compared to that for the ripple carry adder. In most cases, though, a slightly slower version requiring much less hardware, would suffice.

5. CARRY SKIP ADDER

The carry skip, or anticipatory carry, technique may at first appear quite similar to the carry lookahead method for decreasing carry propagation time. The lookahead method, however, determines the true carry into each bit position as an explicit function of preceding positions; the carry skip method uses a form of ripple carry, bypassing (skipping) carries around bits or groups of bits which would only cause the carry to be propagated.

5.1 Basic Skip Circuit

One of the first, and simplest, carry skip circuits was described by Morgan and Jarvis [12] in 1959. They fragmented the adder into 6-bit groups and made provision for bypassing the carry around the groups. Their basic logic is given in Figure 7.

If the condition

$$T_i = X_i \vee Y_i$$

is satisfied at stage i, an incident carry is enabled to pass on to the next stage through a skip gate. If the above condition is satisfied by all stages in a group, a carry incident on the group will be shunted around the group with minimal delay (Figure 7).

In Figure 7 the only true skip occurs when the input carry, C_{in}, is bypassed through the AND at the bottom of the figure. All remaining carries, wherever generated, must ripple through the logic to C_{out} or be absorbed internally. A carry chain of up to 8 delays (using NANDS), in the present case, can exist within one group.

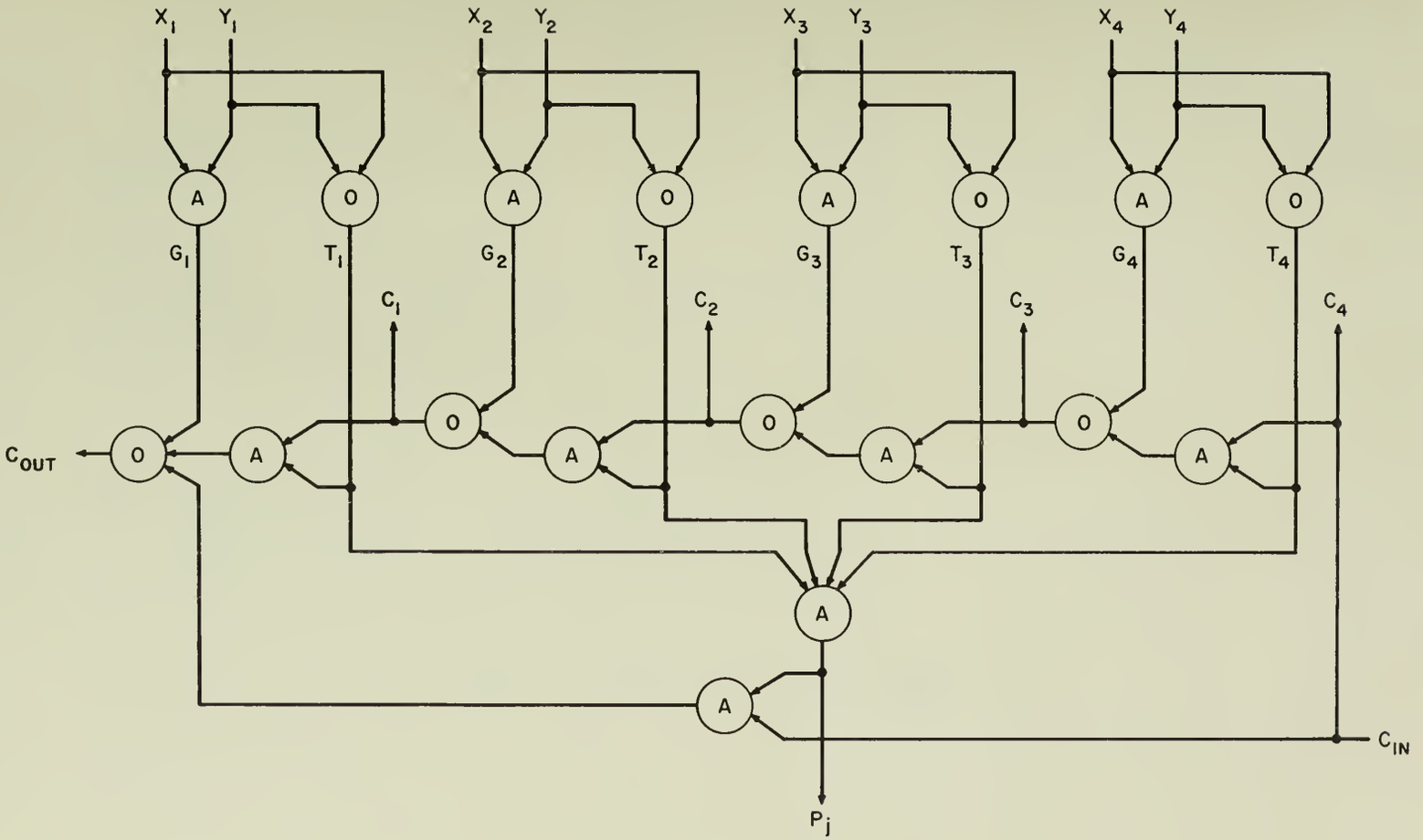


FIGURE 7. ELEMENTARY CARRY SKIP LOGIC FOR 4-BIT ADDER GROUP

The equation for C_{out} of Figure 7 is

$$C_{out} = G_1 \vee T_1 C_1 \vee P C_{in}$$

which is identical to the expression for the ripple carry, except for the $P C_{in}$ term, the carry skip term, where P is the propagation function, equal to $T_1 T_2 T_3 T_4$.

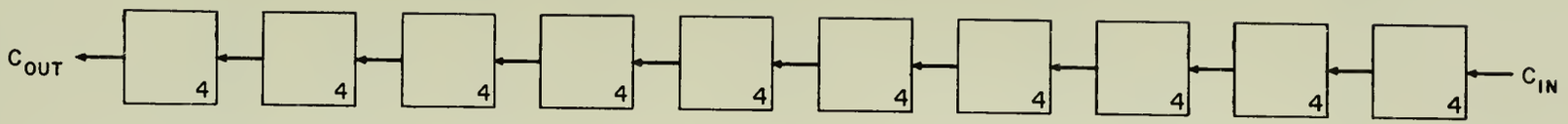
The carry skip group of Figure 7 can be employed in a 40-bit adder like that of Figure 8a. This adder yields a worst-case add time of 40 per cent of that of the ripple carry adder with but a 4 per cent hardware increase (using the adder stage of Figure 11).

5.2 Extension of Carry Skip

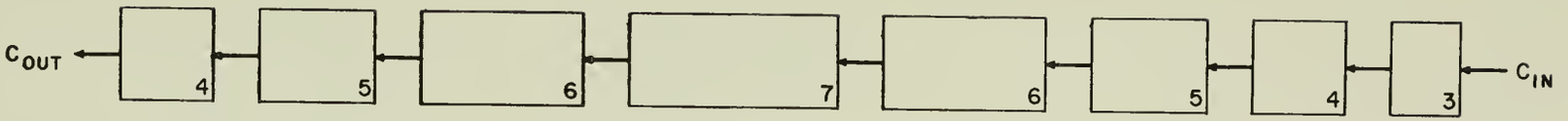
In view of the ideas and logic presented in the previous section, Lehman and Burla [9] posed the following fundamental questions: (1) What is the optimum group size for equal-sized groups for n-bit binary adders? (2) Can the circuit be further speeded up by using unequal-sized groups? (3) Is it advantageous and economical to provide skip gates within or between the groups, or both?

In answer to (1), it was shown [9] that the optimum group size is about n/10 bits for an adder n bits long (n greater than 30). This is the optimum size only if fixed group size is used throughout.

As for (2), unequal groups will speed up the logic. The optimum division is obtained when group sizes increase by one stage (bit) for each group up to the middle group (pair of groups), and then decrease in a similar manner. Thus, the dependence of the carry propagation time on the carry chain length is almost eliminated.



(a) FIXED GROUP SIZE (FIGURE 7)



(b) VARIABLE GROUP SIZE

FIGURE 8. EFFECT ON ADD TIME OF GROUP SIZE IN CARRY SKIP ADDER

Figure 8b gives an example of the adder obtained using unequal groups. Figure 8a shows a 40-bit adder made up of 4-bit groups like the group of Figure 7. The adder of Figure 8a has an add time of 32 delays; that of Figure 8b is 26 delays. Moreover, the faster adder of Figure 8b uses less hardware! It can be concluded that an adder should be composed of fixed length groups of optimum size only if it is not feasible to use groups of varying size.

The skip techniques discussed so far are essentially primitive since only a few of the many possible carry chains have been bypassed. Inter- and intra-group skips may be postulated. By adding a sufficient number of skip circuits, the add time may be reduced as much as desired to a lower limit of eight delays (for a 40-bit adder) -- two to form carries, four to propagate them everywhere, and two to form the final sum.

Figure 9 gives the logic for a 4-bit adder group with skip circuitry capable of sum formation within six delays after X, Y, and C_{in} are presented. C_{out} will be present within four delays. Without any skip gates except the group bypass, the add time is eight delays, and C_{out} formation also takes eight. It is not possible to add another nonredundant skip gate to this group (Figure 9).

As the group size is increased, the time savings becomes more significant when full carry skip logic is employed. A 7-bit group, for example, needs 14 delays without skip gates, and only six delays with all possible nonredundant skip paths to form the output carry.

Groups of groups may also be bypassed. The propagate signals, P_j, output from each group (Figure 7 or 9) are brought together to control the inter-group skips. Again, the idea of increasing the number of groups in a section by one for each section, from the outside to the center, pays off. Figure 10 shows two adders, one using 4-bit groups,

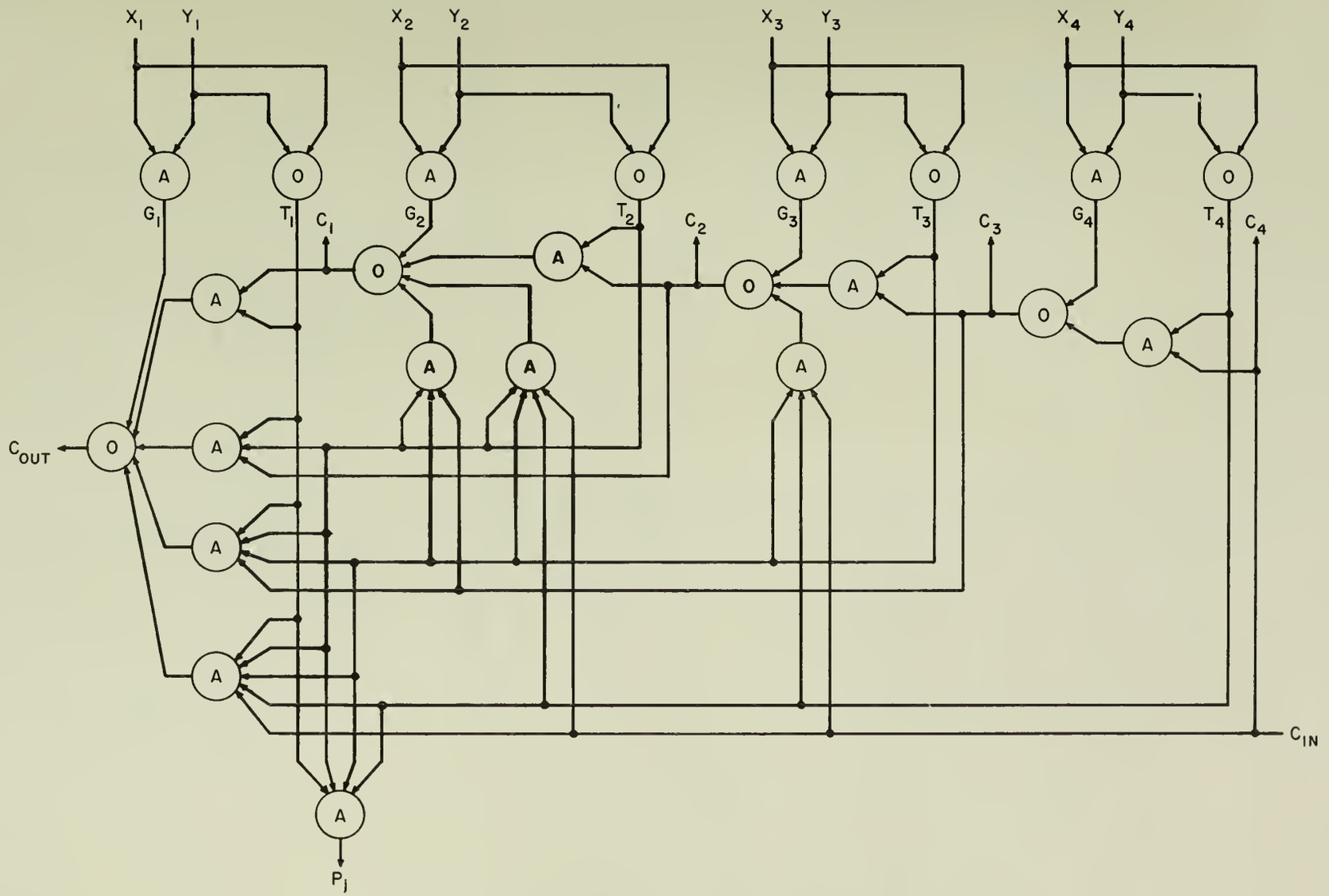


FIGURE 9. 4-BIT ADDER GROUP
WITH FULL CARRY SKIP

the other, 5-bit groups. The 45-bit adder using 5-bit groups and fixed section size adds in 26 delays; the 48-bit adder using 4-bit groups and variable section size, adds in 24 delays by employing the adder logic of Figure 11 in the group logic shown in Figure 7. Even more time savings would be realized if the end groups were shortened and the center group(s) increased in size as outlined above.

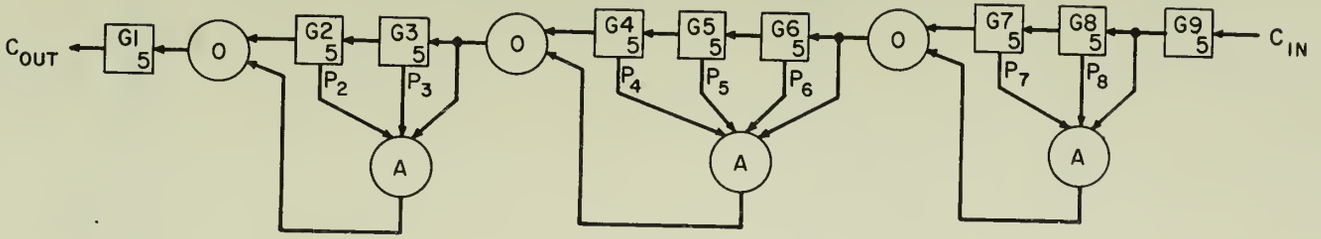
Figure 12 gives an abbreviated form for the logic of a 40-bit adder with full carry bypassing. This is the ultimate speed carry skip adder since all possible skip paths have been provided. Regardless of the adder size, eight delays are required to complete an addition. (The actual adder size, and hence, speed, will be limited by gate capabilities.)

5.3 Conclusion

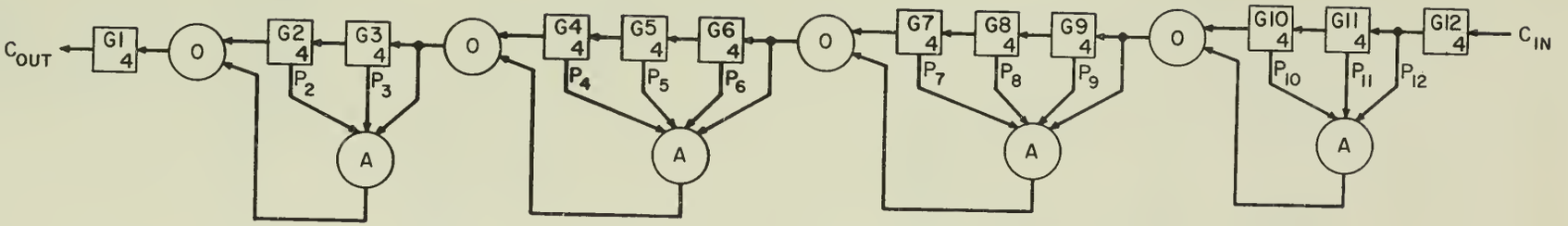
In the design of a carry skip adder, the designer should try several combinations of group size and arrangements of inter- and intra-group skip paths in order to obtain the desired speed at minimum cost. Where the alternative exists, the groups should be made smaller, increasing inter-group skips while decreasing the number of skips internal to the groups.

The adder design should be inspected for redundant carry skip gates. A gate is redundant if it enables an input carry to be present at a bit position before the total time needed for carry propagation has elapsed. Redundant skip gates, if present, may be eliminated with no accompanying degradation in add time.

The carry skip adder offers the logic designer a method by which the add time can be significantly decreased with but little increase in hardware over the ripple carry adder. As the skip paths become more numerous and complex, the logic approaches that of the full carry lookahead adder as the limiting case.



(a)



NOTE: GROUPS ARE SAME AS SHOWN IN FIGURE 7.

(b)

FIGURE 10. 45 AND 48-BIT CARRY-SKIP ADDERS

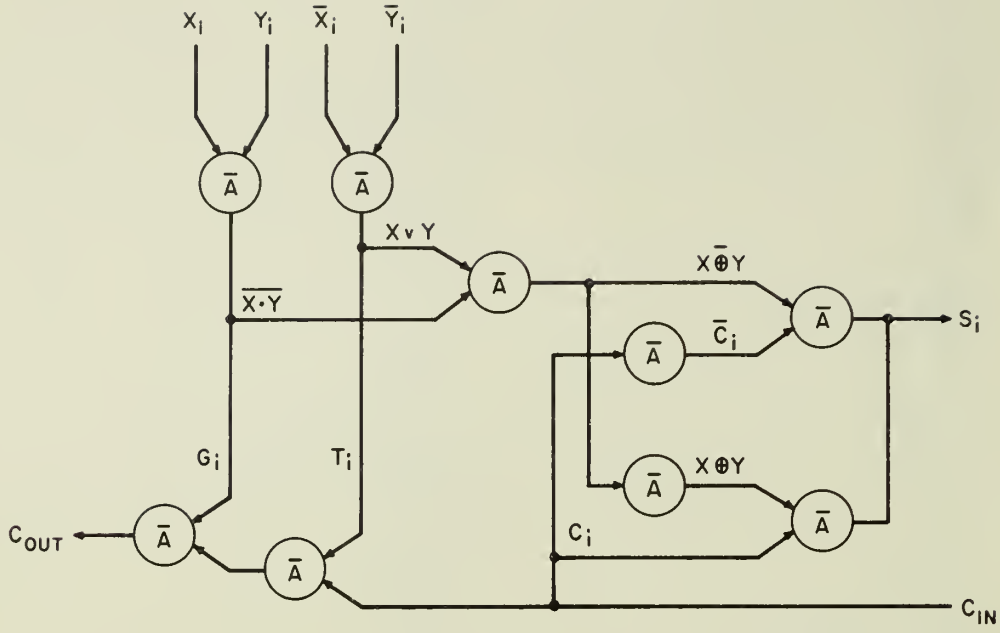


FIGURE 11. ADDER STAGE OF FIGURE 3 MODIFIED FOR USE IN CARRY SKIP ADDER

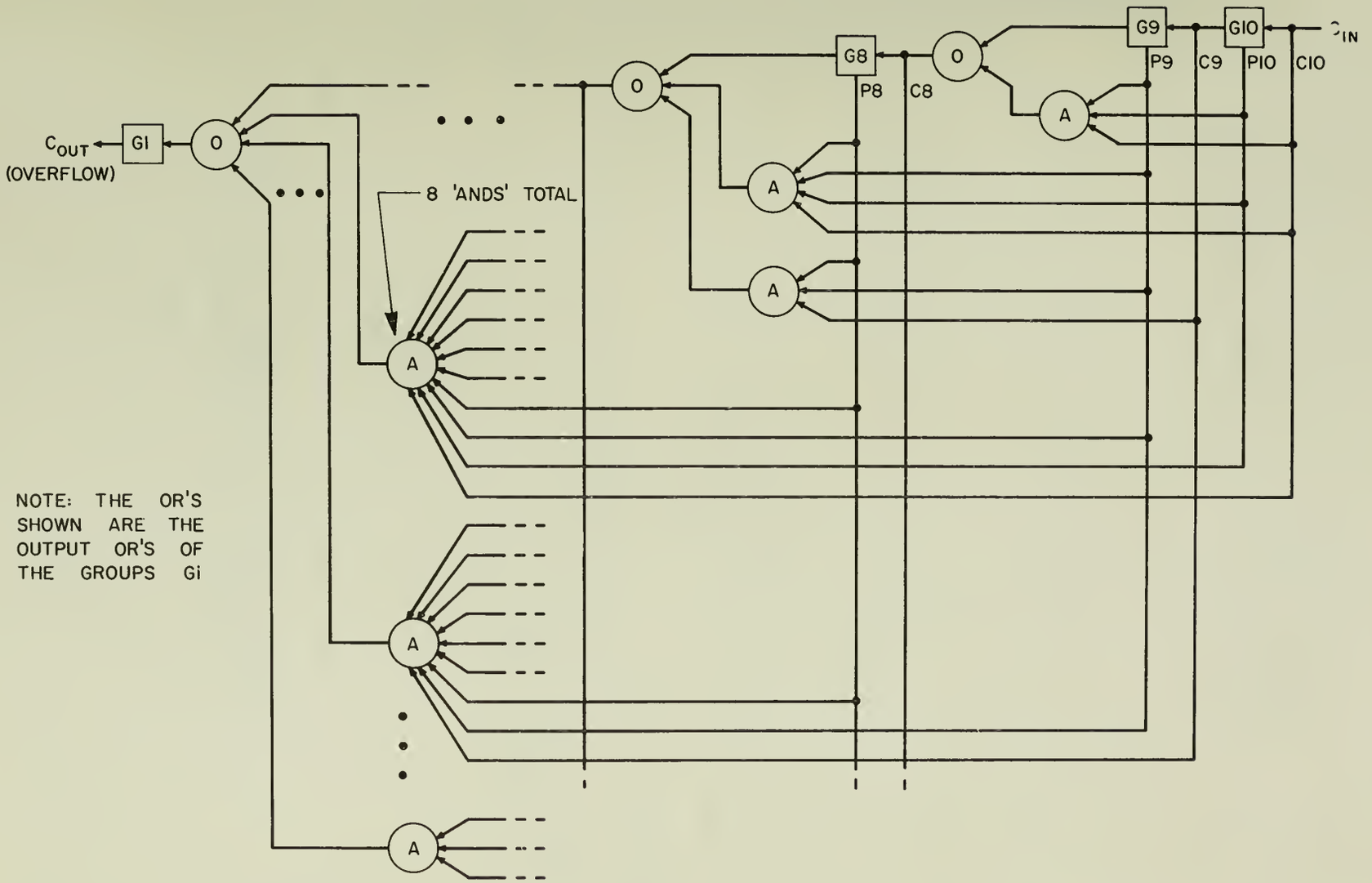


FIGURE 12. ABBREVIATED 40-BIT CARRY SKIP ADDER WITH ALL NONREDUNDANT SKIP PATHS PROVIDED FOR

6. THE CONDITIONAL SUM ADDER

The designs of the previous three adder-types were based, in part, on the philosophy that no action should be taken towards the generation of the true sum bit at any stage until the true carry into that stage is present. The carry lookahead and carry skip logics were means to propagate the carries as fast as possible so that the sum bit could be formed in the shortest time possible (or desired).

The question now arises as to whether the true carry must be present in order for the true sum bit to be generated. Answer: it must. However, in a binary adder, there are but two possible carries into any bit position or group of bits: a '1' or a '0'. So there is no reason not to construct an adder which generates sums subject to the conditions that a carry is and is not input to each stage or group. The true sum can then be selected "later", when the true carry, be it a '1' or a '0', comes along. Hence, the conditional sum or carry select adder results.

6.1 Basic Theory and Logic

Conditional sum addition was first described in 1960 [16]. It is based on the computation of "conditional" sums and carries which result from all possible distributions of input carries for various bit positions. Each bit position of the adder generates a sum bit and a carry bit under the assumption that there is a carry into that position, and another sum and carry bit under the opposite assumption that there is no carry input. Then pairs of conditional sums and carries are combined assuming there is (and is not) a carry into that pair of bits. This process continues until the true sum results. Figure 13 illustrates this process, with a forced input carry, C_n, of '0'. Since the carry input to the low order position is known, the true sum can be formed.

X	0	1	1	0	0	0	1	1	ASSUMED INITIAL CARRY	TIME INTERVAL
Y	0	0	1	0	0	1	1	1		
S_N	0	1	0	0	0	1	0	0	0	τ_1
C_N	0	0	1	0	0	0	1	1		
S_Y	1	0	1	1	1	0	1	*	1	
C_Y	0	1	1	0	0	1	1			
S_N	0	1	0	0	0	1	1	0	0	τ_2
C_N	0		1		0		1			
S_Y	1	0	0	1	1	0			1	
C_Y	0		1		0					
S_N	1	0	0	0	1	0	1	0	0	τ_3
C_N	0				0					
S_Y	1	0	0	1					1	
C_Y	0									
TRUE SUM	1	0	0	0	1	0	1	0		τ_4
TRUE CARRY	0								0	

* CARRY INPUT IS KNOWN TO BE '0' SO THIS DOES NOT NEED TO BE COMPUTED

FIGURE 13. EXAMPLE OF ONE METHOD OF CONDITIONAL SUM ADDITION

The relations are the sums and carries out of each bit position are

$$S_{Ni} = X_i \oplus Y_i$$

$$C_{N(i+1)} = X_i\, Y_i \qquad \text{for '0' carry input}$$

$$S_{Ei} = S_{Ni} \oplus 1 = \bar{S}_{Ni}$$

$$C_{Ei} = X_i \vee Y_i \qquad \text{for '1' carry input}$$

where the subscripts N and E indicate no carry input and an input carry (exists), respectively.

The first row sums and carries are generated according to the above relations. The second (τ_2) and succeeding rows are formed similarly with the right-hand member of each pair (group) using an assumed input carry (no carry) and the left-hand member(s) using the output carry of the right-hand member(s) from the row above. The process continues until all stages have been paired and re-paired until the true sum is output. The details will become clearer as the reader studies Figure 13. Sklansky [17] has a more detailed explanation with illustrations.

6.2 Improved Conditional Sum Logic

The above method of generating conditional sums is not particularly attractive for use in a high-speed system since it is limited in speed to two delays per row. A 40-bit adder would need, therefore, about 14 delays to complete an addition. Also, the logical structure is, perhaps, somewhat more complex than is desirable.

The adder of Section 6.1 is further limited in size and speed by the fanout of the logical elements used. A version which does not need high fanout logic, though it may be slower than that of Section 6.1, employs the ripple carry adder of Section 2.

In this case, the adder system is divided into groups of from 3 to 6 bits each. Each group contains two identical ripple carry adder sections. If the groups are assumed to be 4 bits long, then one ripple carry adder section forms the sums S_{E1} through S_{E4} (input carry assumed '1'), and the other, similar section forms S_{N1} through S_{N4} (input carry assumed '0').

Simultaneously with carry propagation within the group, two output carries, C_E and C_N, are produced using lookahead principles. While carries are rippling through the adders internal to each group, the conditional carries are rippling between groups, forming the true input carries to each group. The conditional output carries from every group are combined with carries from all preceding groups. The forced carry, C_n, is the input to the low order group.

These true input carries and their complements, in turn, are routed to the output select gate which selects the proper sum bit group, S_E or S_N, and causes it to be gated out of the adder system via the output select gate. Figure 14 gives the logic for one of the 4-bit groups described above.

The true carries are formed according to

$$C_i = C_{Ni} \vee C_{Ei}\, C_{i-4}$$

$$C_{i+4} = C_{N(i+4)} \vee C_{E(i+4)}\, C_i, \text{ etc.}$$

Two versions of the conditional sum adder logic are given: ripple through and lookahead (Figure 15). The ripple carry propagate realization is slower, but requires a lower fanin/fanout from the elements than does the lookahead version, which also requires more hardware.

The basic adder group (4 bits) for the conditional sum adder is shown in Figure 14. Complete circuit duplication is not necessary since both the S_E and S_N adder sections utilize the same primary functions:

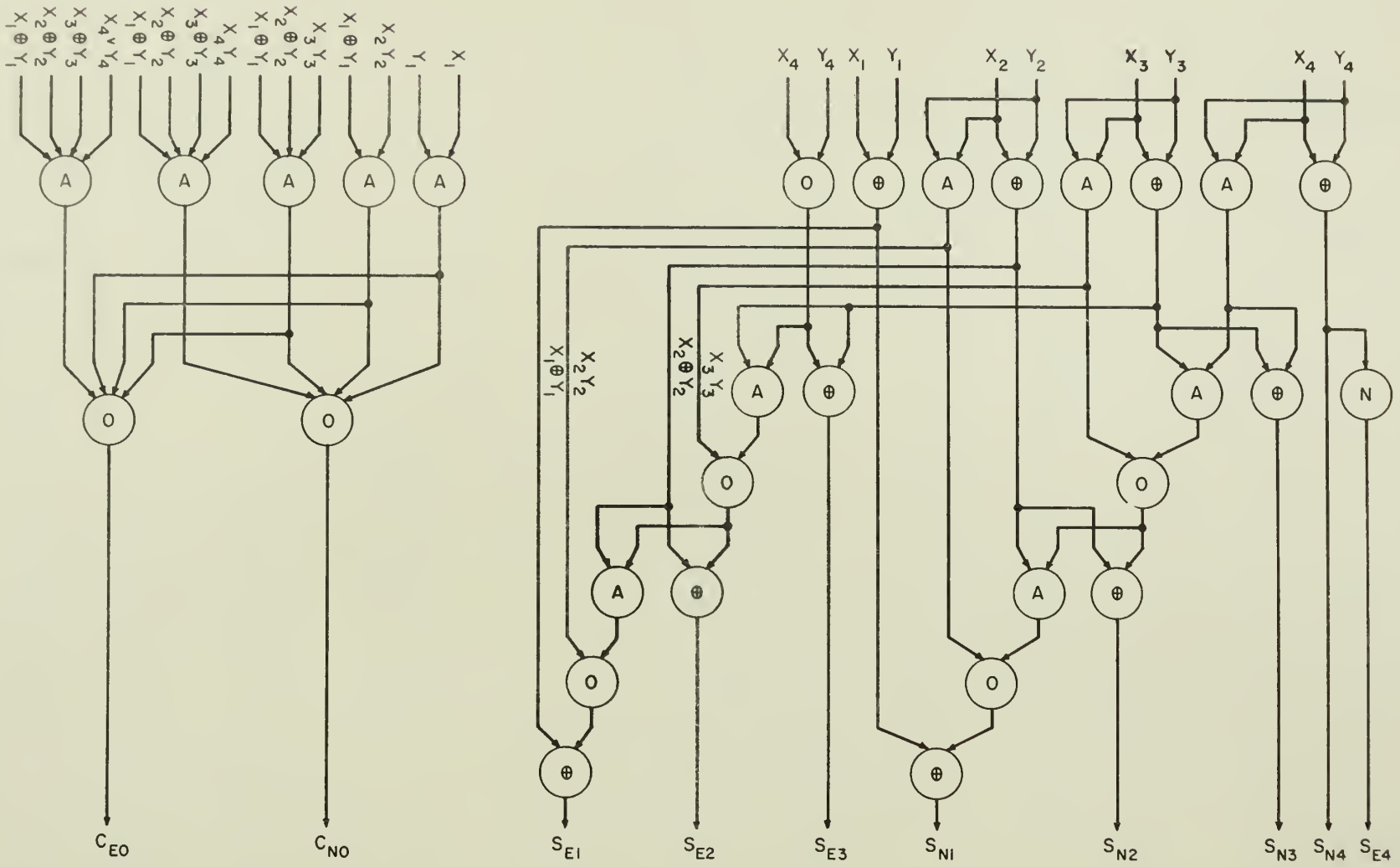
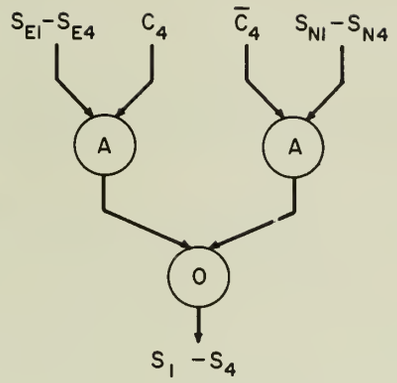


FIGURE 14. 4-BIT CONDITIONAL SUM ADDER SECTION USING RIPPLE CARRY LOGIC



$C_i, \bar{C}_i$ ARE COMMON TO ALL BITS IN A GROUP

FIGURE 15a. OUTPUT SELECT GATE (OG)– ONE PER BIT

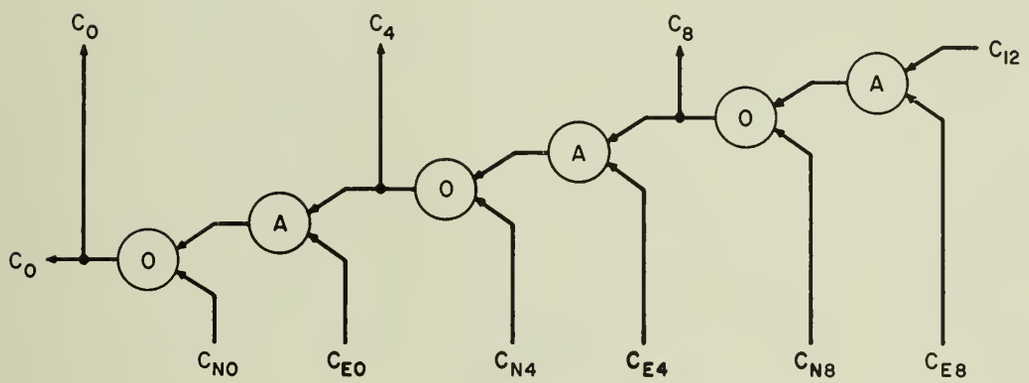


FIGURE 15b. RIPPLE THROUGH TRUE CARRY SELECTION LOGIC

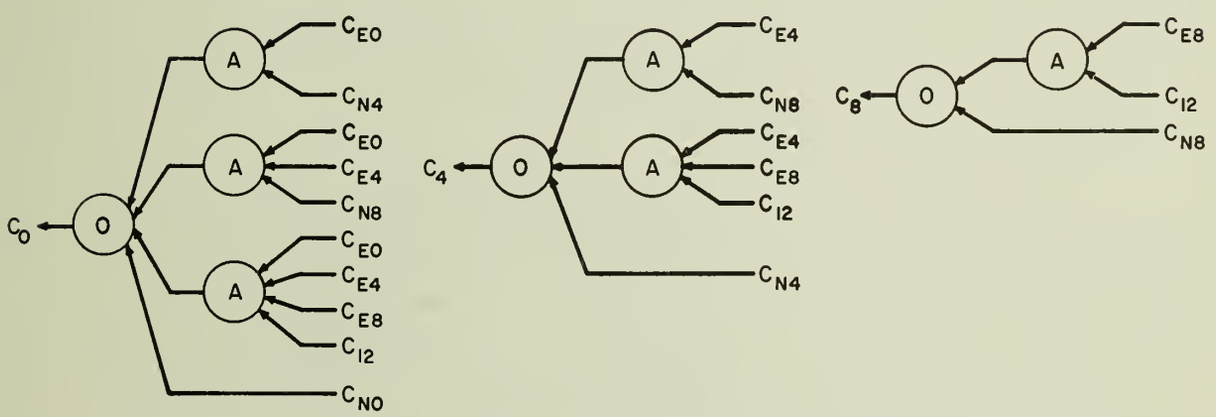


FIGURE 15c. LOOKAHEAD TRUE CARRY SELECTION LOGIC

$X \oplus Y$ and $X \cdot Y$. In this ripple carry version, the carry out of stage 4 is given by:

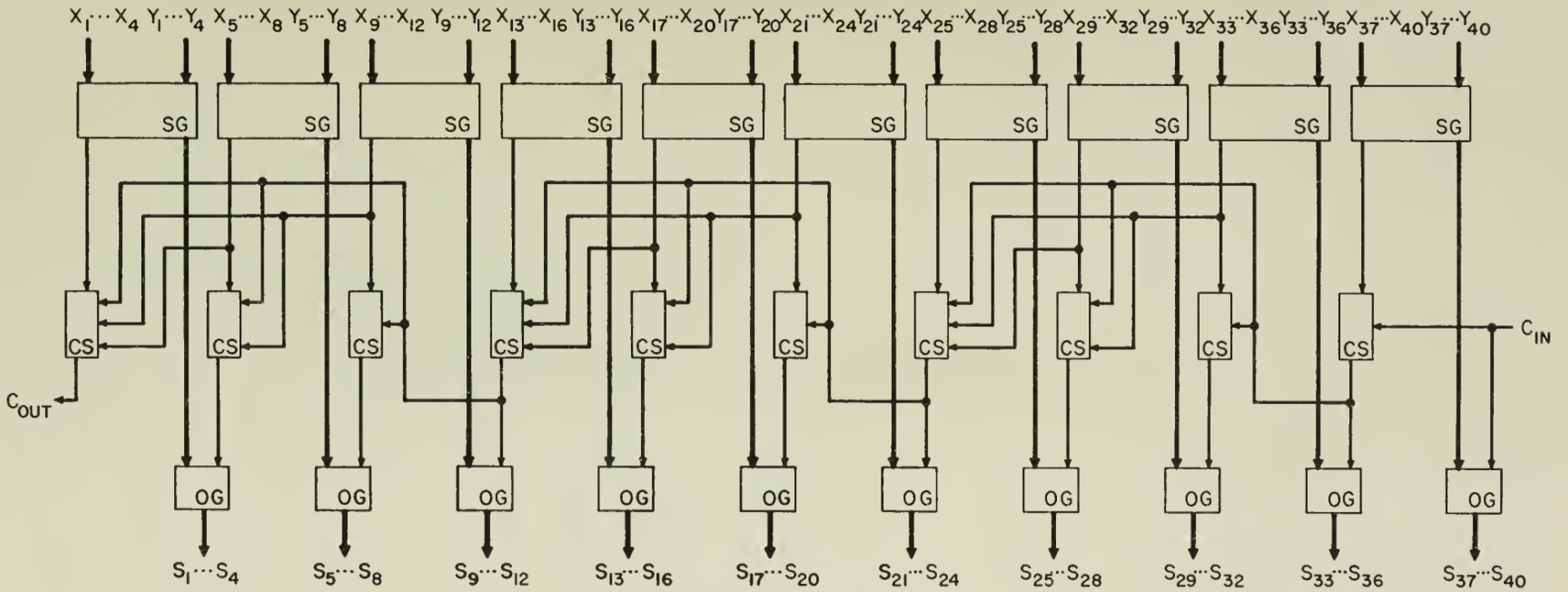
$C_{E3} = X_4 \vee Y_4$ for an assumed input carry of '1' (S_Y generation)

$C_{N3} = X_4 \cdot Y_4$ for an assumed input carry of '0' (S_N generation)

Figure 16 presents the block logic for a 40-bit carry lookahead conditional sum adder. This particular realization uses ripple carry in the sum generation blocks and carry lookahead over sections of three groups each for true carry generation. If straight ripple carry logic were used (Figure 15b) for true carry selection, 25 delays would be necessary to generate and propagate the carries. The conditional sums, however, are all ready after only eight delays. If the carry lookahead logic (Figure 15c) is employed rather than ripple carry logic (retaining carry ripple between sections), the true carry propagation time can be reduced to 13 delays. Two further delays are necessary for sum selection in the output gates (OG in Figure 15a).

Using the ripple and lookahead carry logic as described above in the adder of Figure 16, an adder is obtained which will complete an addition in 15 delays. However, a 75 per cent increase in hardware over the ripple carry adder is needed.

One of the "secrets" of conditional sum adder logic is the matching of the sum production and true carry propagation times. The example given in Figure 16 did not achieve this match, but a closer match could be realized by utilizing full-carry lookahead for the carry propagation logic in place of partial lookahead.


$X_1 \cdots X_4$ $Y_1 \cdots Y_4$ $X_5 \cdots X_8$ $Y_5 \cdots Y_8$ $X_9 \cdots X_{12}$ $Y_9 \cdots Y_{12}$ $X_{13} \cdots X_{16}$ $Y_{13} \cdots Y_{16}$ $X_{17} \cdots X_{20}$ $Y_{17} \cdots Y_{20}$ $X_{21} \cdots X_{24}$ $Y_{21} \cdots Y_{24}$ $X_{25} \cdots X_{28}$ $Y_{25} \cdots Y_{28}$ $X_{29} \cdots X_{32}$ $Y_{29} \cdots Y_{32}$ $X_{33} \cdots X_{36}$ $Y_{33} \cdots Y_{36}$ $X_{37} \cdots X_{40}$ $Y_{37} \cdots Y_{40}$
SG
CS
OG
C_{OUT}
C_{IN}
$S_1 \cdots S_4$ $S_5 \cdots S_8$ $S_9 \cdots S_{12}$ $S_{13} \cdots S_{16}$ $S_{17} \cdots S_{20}$ $S_{21} \cdots S_{24}$ $S_{25} \cdots S_{28}$ $S_{29} \cdots S_{32}$ $S_{33} \cdots S_{36}$ $S_{37} \cdots S_{40}$

SG = SUM GENERATOR (SEE FIGURE 14)
CS = CARRY SELECTOR (SEE FIGURE 15)
OG = OUTPUT GATES (SEE FIGURE 15)

FIGURE 16. 40-BIT CONDITIONAL SUM ADDER USING LOOKAHEAD CARRY SELECT (CS) LOGIC

6.3 Further Speed Increases

It was seen above how the use of carry lookahead in the true carry propagation logic reduced the add time from 25 to 15 delays. But the version given there uses ripple carry in the 4-bit sum groups. One could therefore suppose that introducing full carry lookahead into the conditional sum generating groups and full carry lookahead into the true carry propagation path would yield a further increase in speed. This is the case; however, the increase in speed obtained may not justify the extra hardware needed. Figure 17 presents the logic of a 4-bit conditional sum group with full carry lookahead.

The sums and output carries from this group are generated according to the equations

$$S_{N4} = X_4 \oplus Y_4$$
$$S_{E4} = X_4 \bar{\oplus} Y_4 = \overline{S_{N4}}$$
$$S_{N3} = X_3 \oplus Y_3 \oplus G_4$$
$$S_{E3} = X_3 \oplus Y_3 \oplus T_4'$$
$$S_{N2} = X_2 \oplus Y_2 \oplus (G_3 \vee T_3\, G_4)$$
$$S_{E2} = X_2 \oplus Y_2 \oplus (G_3 \vee T_3\, T_4')$$
$$S_{N1} = X_1 \oplus Y_1 \oplus (G_2 \vee T_2\, G_3 \vee T_2\, T_3\, G_4)$$
$$S_{E1} = X_i \oplus Y_1 \oplus (G_2 \vee T_2\, G_3 \vee T_2\, T_3\, T_4')$$
$$C_{N0} = G_1 \vee T_1\, G_2 \vee T_1\, T_2\, G_3 \vee T_1\, T_2\, T_3\, G_4$$
$$C_{E0} = G_1 \vee T_1\, G_2 \vee T_1\, T_2\, G_3 \vee T_1\, T_2\, T_3\, T_4'$$

where

$$G_i = X_i \cdot Y_i$$
$$T_i = X_i \oplus Y_i$$
$$T_4' = X_4 \vee Y_4$$

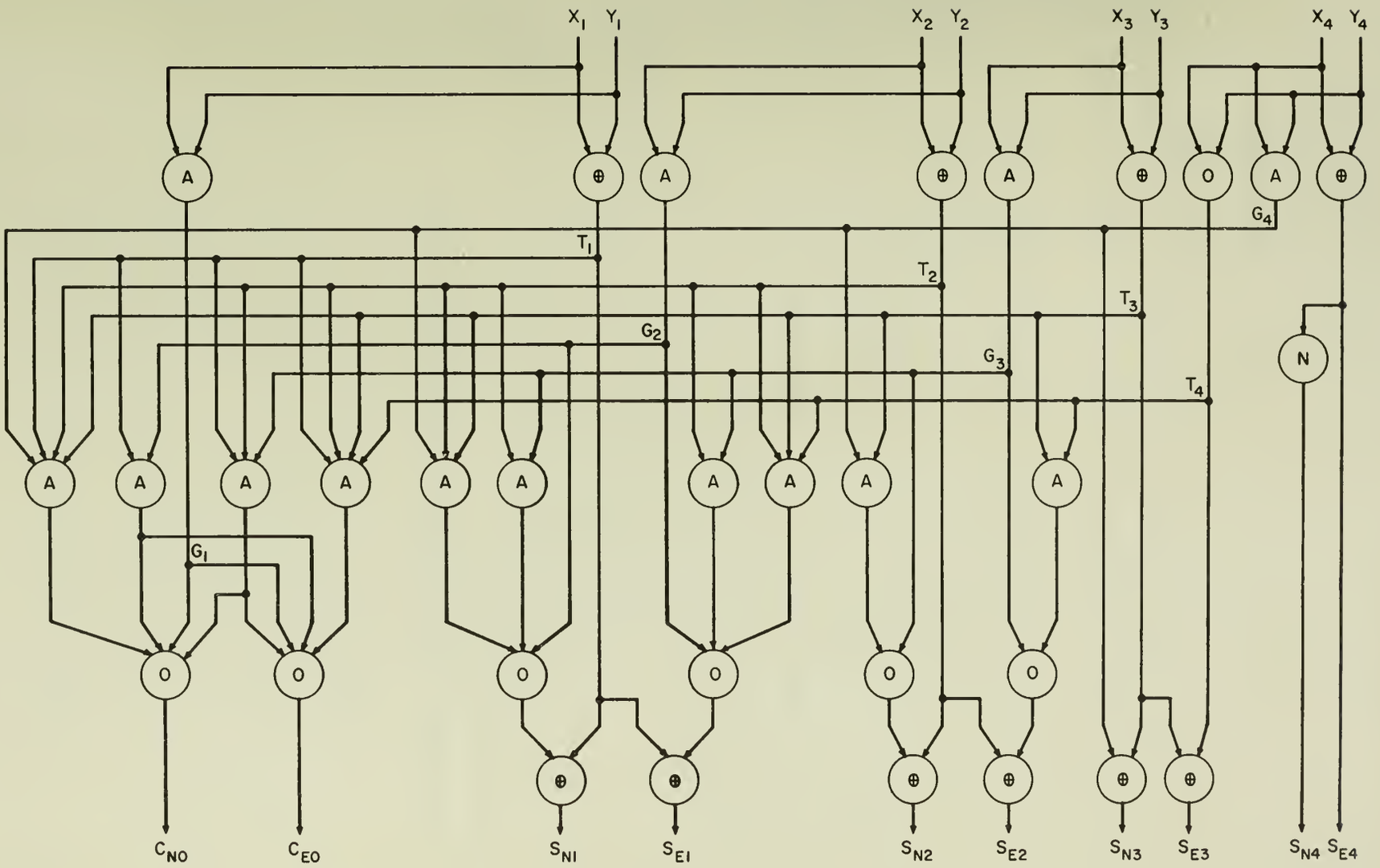


FIGURE 17. 4-BIT GROUP OF CONDITIONAL SUM ADDER WITH FULL CARRY LOOKAHEAD

If NAND logic is assumed, four delays are necessary to form the conditional output carries, C_{Ei} and C_{Ni}, and two more are needed to propagate them to the output gates and form the true carries everywhere in the adder. Sum generation takes five delays. After two further delays needed to combine the sums and carries in the output select gates, the addition is complete within eight delays. The price for this speed is high compared to other addition methods: an 88 per cent increase in hardware over the ripple carry adder.

The conditional sum adder with full-carry lookahead is not the same as a single sum adder with full lookahead logic. The lookahead adder waits for the true carry to arrive before it outputs a sum bit. The conditional sum adder does not concern itself with carries when forming the sum bits -- sums and carries are formed simultaneously, and on the basis of the decoded carries, the proper set of sum bits is selected.

A theoretical delay total of eight is not the ultimate in speed for a conditional sum adder. By changing the form of the expressions for the carry into each bit position, and by using some AND-OR networks in place of NAND cascades (where fanout conditions permit), the add time can be cut to four to six delays.

The original expressions for the input carries were

$$C_{N3} = G_3$$

$$C_{E3} = T_4' = G_4 \vee T_4$$

.

.

.

$$C_{N0} = G_1 \vee T_1 G_2 \vee T_1 T_2 G_3 \vee T_1 T_2 T_3 G_4$$

$$C_{E0} = G_1 \vee T_1 G_2 \vee T_1 T_2 G_3 \vee T_1 T_2 T_3 T_4'$$

If the complements of the carries are formed, this results in the following expressions

$$\overline{C_{N3}} = \overline{G_4}$$

$$\overline{C_{E3}} = \overline{G_4}\ \overline{T_4}$$

.

.

.

$$\overline{C_{NO}} = \overline{G_1}\ \overline{T_1} \vee \overline{G_1}\ \overline{G_2}\ \overline{T_2} \vee \overline{G_1}\ \overline{G_2}\ \overline{G_3}\ \overline{T_3} \vee \overline{G_1}\ \overline{G_2}\ \overline{G_3}\ \overline{G_4}$$

$$\overline{C_{EO}} = \overline{G_1}\ \overline{T_1} \vee \overline{G_1}\ \overline{G_2}\ \overline{T_2} \vee \overline{G_1}\ \overline{G_2}\ \overline{G_3}\ \overline{T_3} \vee \overline{G_1}\ \overline{G_2}\ \overline{G_3}\ \overline{G_4}\ \overline{T_4}$$

which are simpler than the original expressions (one fewer term than before) and faster in formation, since, using NANDS, the function $G = XY = \overline{\overline{XY}}$ $= \overline{\overline{X} \vee \overline{Y}}$ takes two NANDS to form, but $\overline{G}$ needs only one. Also, an AND-OR-INVERT cascade is often faster than an INVERT-INVERT cascade. Another two delays can be saved in the output gate, by using an AND-OR complex in place of the NANDS, and one in the true carry lookahead/propagation logic.

In its ultimate form, the conditional sum adder with full-carry lookahead needs four NAND delays plus three AND-OR delays to form and output the sum. The speed-limiting factor is the driver fanout and gate fanin capacity, since as the adder size increases, these limits are quickly attained. For a driver fanout capability of 18, 24 bits is the maximum length of the adder which can be constructed within the framework outlined above. However, the true carry out of the last stage can be used as the input to another section of up to 24 bits, with one additional delay. The use of the AND-OR in place of NANDS will realize a hardware savings and will probably give a speed increase as a bonus.

6.4 Conditional Sum Adder Summary

The conditional sum adder is the fastest of the fixed-time adders; it is also the most expensive in terms of the number of components used. Its add time is independent of number length to the limits of hardware fanout capability. The conditional sum addition method is a powerful scheme which should not be overlooked by the designer.

7. THE ASYNCHRONOUS ADDER

This section introduces, and concludes, the discussion of asynchronous (speed-independent or variable-time) adders. Previous sections have covered fixed-time addition methods, so designated because a fixed interval of time is allotted for each addition. In the fixed-time adder, even if the addition is completed long before the allotted time has elapsed, the full time is taken to allow for the worst-case conditions. The variable-time adder takes advantage of the fact that, in a ripple carry adder in particular, the addition may be completed before the worst-case add time elapses.

7.1 Principles of Operation

The worst-case add time condition in a ripple carry adder occurs when a carry injected into the lowest-order position ripples completely through to the highest-order position. A fixed-time adder must allow for this case on every addition even though the average carry propagation path length is about $\log_2 n$ positions, where n is the adder length. For example, in a 40-bit adder, the average carry length is 5.6 stages.

This carry path length is seen to be short compared to the adder length. The asynchronous adder uses this fact and achieves its speed by detecting the end of carry propagation. When the carry ripple completion is detected, the adder outputs a "done" signal signifying the completion. One of the problems associated with this method is in insuring that the "done" signal is not transmitted until the addition is truly complete. Also, the "done" signal should be removed if any of the inputs is changed during an addition.

The above problems can be overcome if two carry propagation lines, rather than one, are used -- one for the 'one' or true carry signal; the other for the 'zero' or no carry signal. These lines both rest in the '0' state.

When an addition is performed, "1's" will propagate down the 'one' and 'zero' carry lines according to the expressions:

$$C_i^1 = X_i\, Y_i \vee (X_i \vee Y_i)\, C_{i+1}$$

$$C_i^0 = \overline{X}_i\, \overline{Y}_i \vee (\overline{X}_i \vee \overline{Y}_i)\, \overline{C}_{i+1}$$

where C_i^1 is the 'one' carry and C_i^0 the 'zero' carry out of stage i. The expression for the 'done' (addition complete) or reply signal may be written as

$$R = (C_1^1 \vee C_1^0)(C_2^1 \vee C_2^0)(\ldots)(C_n^1 \vee C_n^0)$$

When the addition is complete, exactly one of the two lines out of every stage will be '1'. If the state of all C_i^1's and C_i^0's is sensed, the addition will be finished when exactly one of each pair

$$\{(C_i^1,\ C_i^0) : i = 1,\ 2,\ \ldots,\ n\}$$

is simultaneously in the '1' state. The C_i^0 signal may be thought of as meaning "no true carry can be propagated through this stage now, nor will one originate here."

7.2 Asynchronous Carry Logic

The truth table of Figure 18a will enable the determination of the carry logic. The carry out is determined by C_{in} or by X_i and Y_i. The carry logic used in the fixed-time ripple carry adder is given in Figure 18b. This circuit is incapable of providing a "done" signal because there is only one carry path, and an input of $X_i = Y_i$ = '0' will cause an output of zero, which is exactly what is not wanted.

C_{IN}	X	Y	C_{OUT}	
0	0	1	0	
0	1	0	0	DETERMINED
1	0	1	1	BY C_{IN}
1	1	0	1	
0	0	0	0	DETERMINED
1	0	0	0	ONLY BY
0	1	1	1	X & Y-
1	1	1	1	NOT C_{IN}

FIGURE 18a. TRUTH TABLE FOR ASYNCHRONOUS ADDER CARRY DETERMINATION

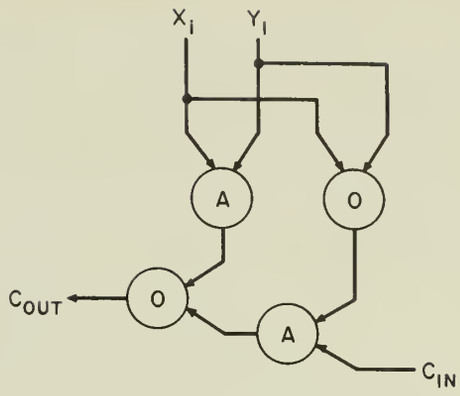


FIGURE 18b. STANDARD RIPPLE CARRY LOGIC

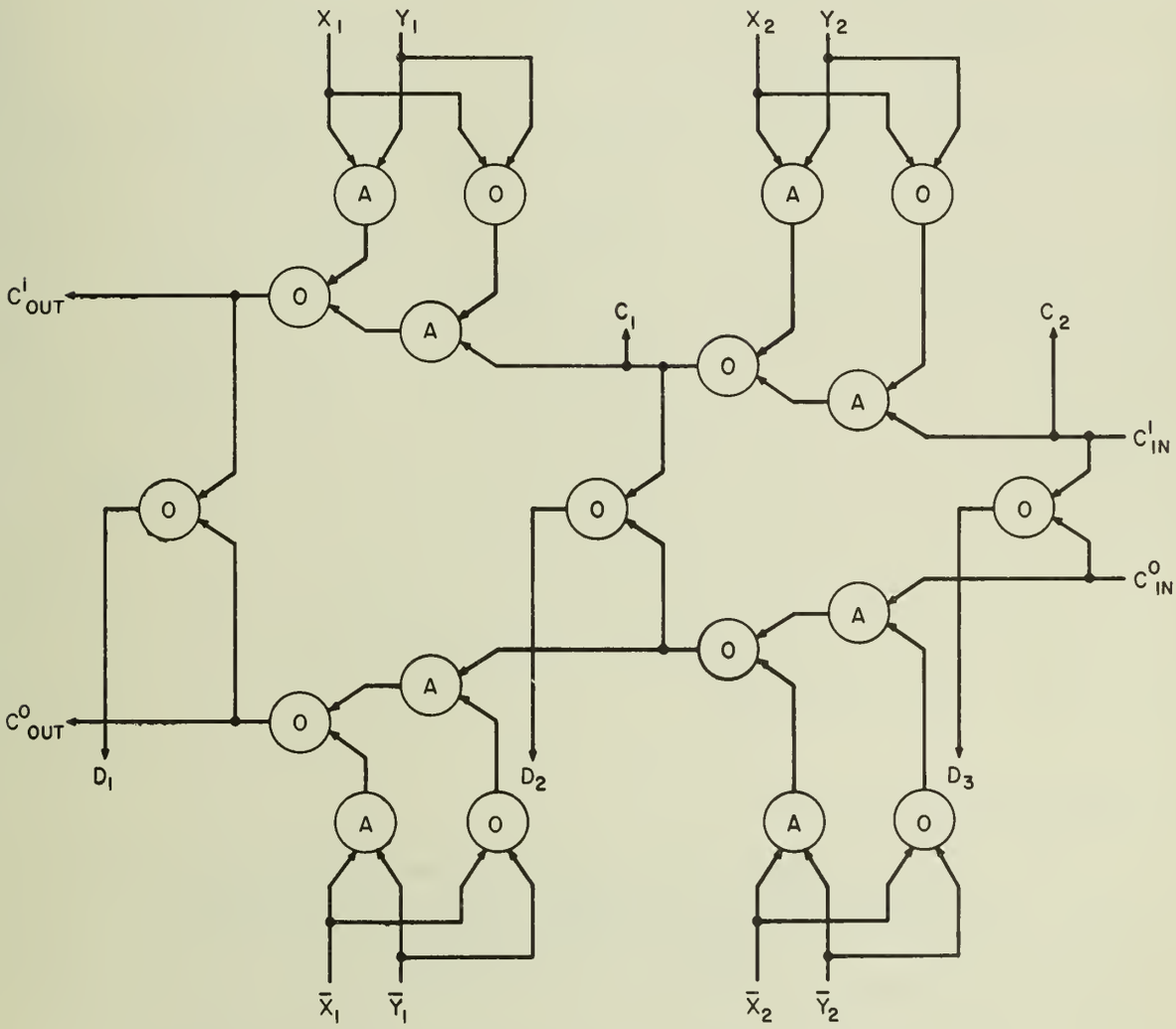


FIGURE 18c. LOGIC FOR CARRY PROPAGATION IN ASYNCHRONOUS ADDER

Figure 18c shows a carry logic which is self-timing since it provides both a 'one'- and a 'zero'- carry path. Both carry paths rest at '0' at the start of an addition. When X and Y are input, one of two conditions is impressed on each carry line at each stage:

(1) A 'one' or a 'zero' carry is injected and the 'zero' or 'one' carry path is blocked, corresponding to $X_i \cdot Y_i$ and $\overline{X}_i \cdot \overline{Y}_i$ inputs respectively; or

(2) An input 'one' or 'zero' carry is allowed to propagate through the stage, corresponding to $X_i \vee Y_i$ and $X_i \vee Y_i$ inputs respectively.

The input 'one' and 'zero' carries propagate along their corresponding paths until they are blocked by a carry injected into the opposite path. After propagating through $\log_2 n$ stages on the average, the carry sequence is finished, and a '1' is output from every end-condition OR circuit (D_1, ... D_n; Figure 18c) to the n-input carry completion AND gate. When all inputs to this AND go to '1', the output follows, signalling the completion of the addition. (All carry sequences start at the same time.)

If any input changes state during the addition, the carry paths will be affected and the carry completion signal will fall to '0' until equilibrium is again established.

7.3 Self-Timing Adder Stage

The adder of Figure 11 may be used in the variable-time adder with little modification. Figure 19 presents the logic, using NANDS, for one stage of the self-timing adder. It is identical to that of the ripple carry adder except for the addition of three NANDS per stage and the n-input (for an n-bit adder) NAND for carry detection.

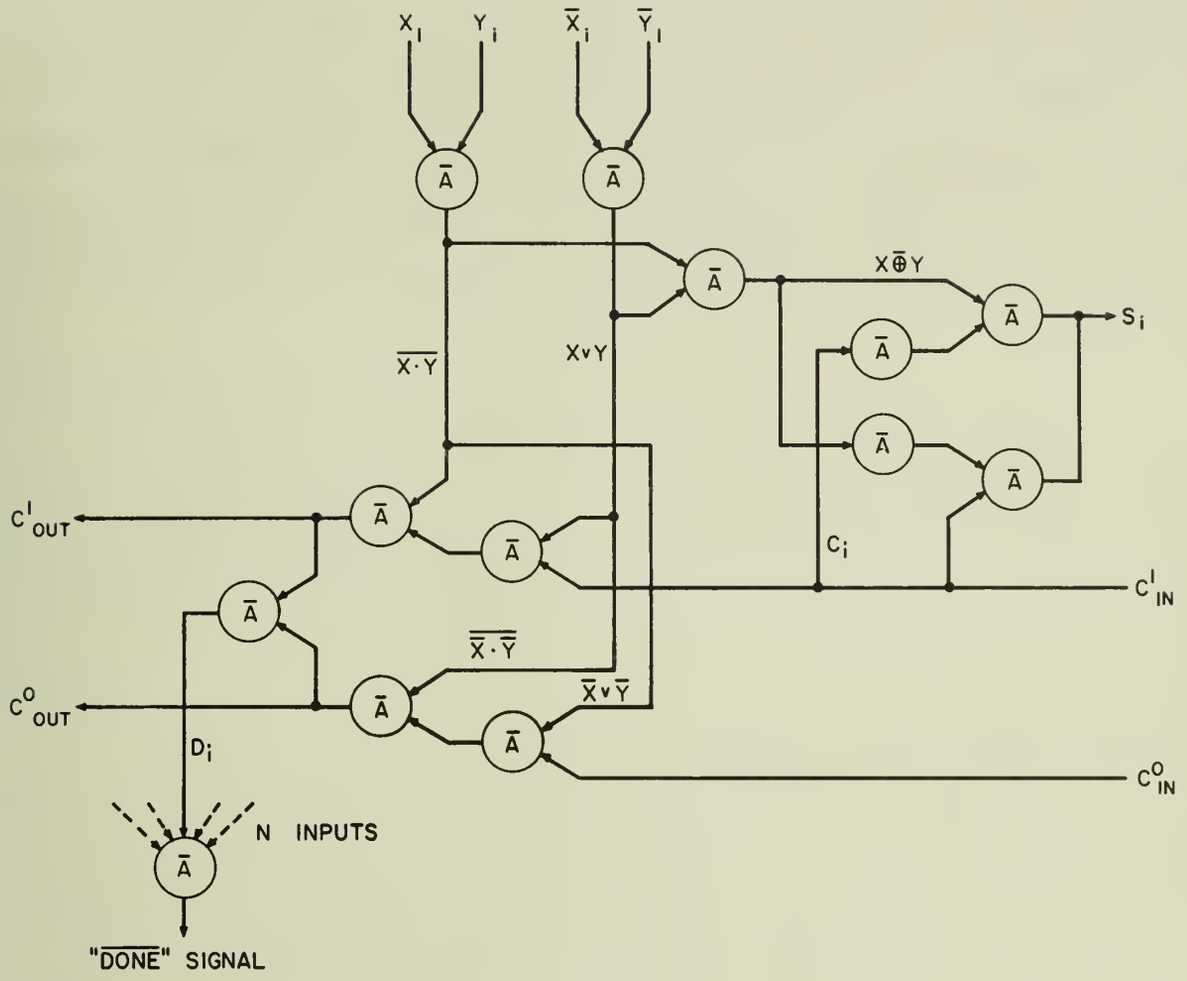


FIGURE 19. ONE STAGE OF ASYNCHRONOUS ADDER USING NANDS

In this example, the carry lines rest at '1' rather than '0' as described above. This is good since this enables the NAND connected to the C^1 and C^0 lines to function as a negative OR. When one of its inputs goes to '0', signifying a true carry on the other line, the NAND's output will rise to '1'. When the outputs of all such NANDS are '1', the output of the n-input NAND will fall to '0', thus indicating the completion of the addition.

A 40-bit self-timing adder may be constructed using about 34 per cent more NANDS than the ripple carry adder. Its speed of operation will vary between a minimum of four delays to a maximum of 82 as compared with 80 delays for a 40-bit ripple carry adder. The average time required will be about 16 delays.

7.4 Variable-Time Adder Summary

Nothing in general may be said about whether a fixed- or variable-time adder is better. The choice of one type or the other depends on the features of the system in which the adder is to be used. If the system will allow a wide spread between average and maximum operation times, the asynchronous adder may suffice. If, however, high-speed operation is essential, or strict timing delays required, then the choice of a fixed-time adder is indicated. As one might think, the asynchronous adder is not too suitable for use in a synchronous machine since it operates as "fast" or as "slow" as necessary, sending out a "done" signal whenever the addition is complete.

8. SEPARATE CARRY-STORAGE ADDER

The adder to be discussed here is not a true adder in the sense of those of Sections 2 through 7. This is a "pseudo-adder," forming partial sums and carries from three inputs: X, Y, and the previous carries, and storing the output sum and carries separately. No attempt is made to propagate carries through the adder -- the carries are only absorbed one stage higher than the stage in which they were generated.

8.1 Principles of Operation

The purpose of the carry-storage adder is to form a pseudo-sum and carry as quickly as possible (in a fixed-time) and then store them in separate registers. Figure 20a shows the register arrangement in which an "ordinary" adder would function. Figure 20b gives the layout for a carry-storage adder system.

In ordinary adders, the contents of the X and Y registers are added one way or another and the sum is placed in the S-register. In the carry-storage adder, the contents of the X and Y registers are combined on the first cycle, the pseudo-sum is placed in the S-register, and the carries (b_i) out of each position are stored in the B register. The pseudo-sum and carries are then gated to the X and C registers (they may also be shifted left or right in transit). On the second and succeeding cycles, the contents of the X, Y, and C registers are combined to form new pseudo-sums and carries. During the final addition, the carries must be propagated through the adder in order to produce the true sum in S. For this carry assimilation, any of the methods of Sections 2 through 7 may be used. The carry-storage adder by itself does not have the power to assimilate carries; external circuitry is required for this.

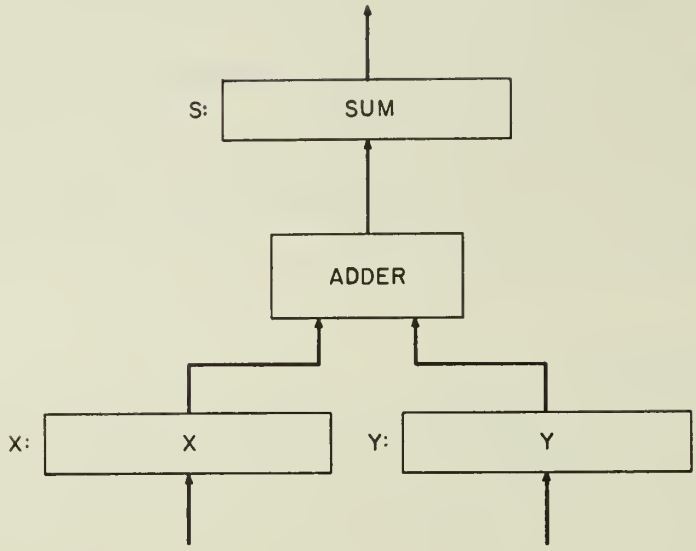


FIGURE 20a. "ORDINARY" ADDER
REGISTER ARRANGEMENT

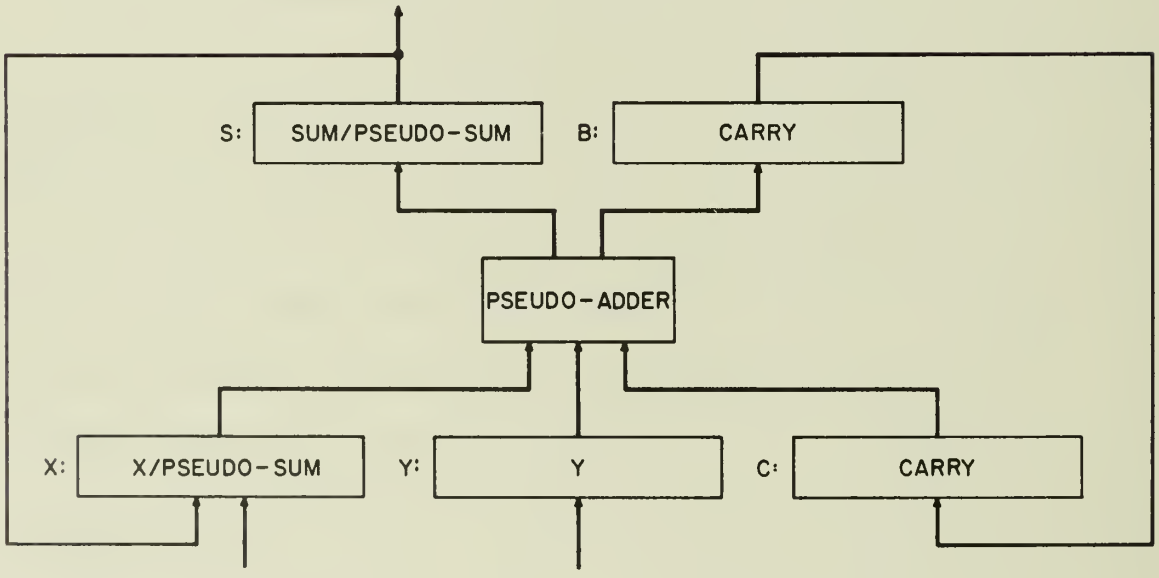


FIGURE 20b. CARRY-STORAGE "PSEUDO-ADDER"
REGISTER LAYOUT

It has been shown [13], that additions, multiplications, etc. can be performed in a consistent fashion using separate stored carries.

8.2 Adder Logic

The adder itself is exactly the same as the ripple carry adder, with but a single modification. The carry propagation chain is broken at the output of each bit position so that the carry produced there might be stored.

There are two places at which the carry chain may be cut: points A and B in Figure 21a. Although the choice is not obvious, the following reasoning supports the choice of point B. Let

$$U_i = X_i \oplus Y_i$$

$$K_i = X_{i+1} \cdot Y_{i+1} \vee b_i$$

where K_i is the carry into bit position i and b_i is defined below.

Then

$$S_i = U_i \oplus K_i = (X_i \oplus Y_i) \cdot b_i \vee X_i \cdot Y_i$$

and

$$b_{i-1} = U_i \cdot K_i \quad = X_i \cdot Y_i$$

where S_i is the pseudo-sum digit and b_{i-1} is part of the carry out of stage i. Note that

$$S_i \cdot b_{i-1} = (U_i \oplus K_i) \cdot U_i \cdot K_i = 0$$

and

$$X_i \cdot Y_i \cdot b_{i-1} = 0$$

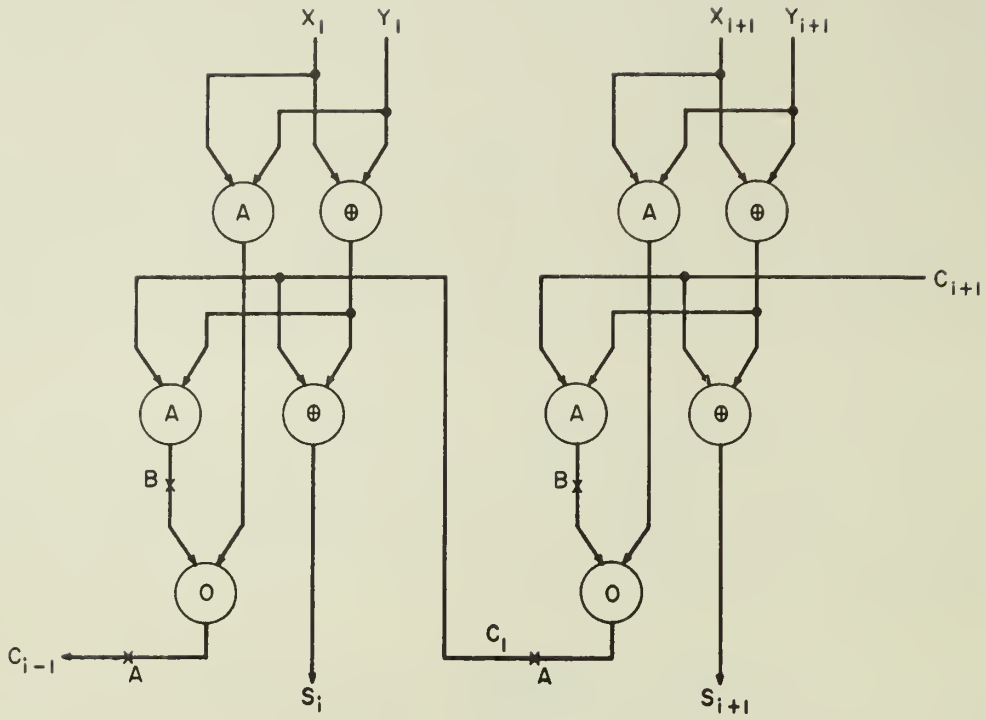


FIGURE 21a. LOGICAL STRUCTURE OF CONVENTIONAL ADDER

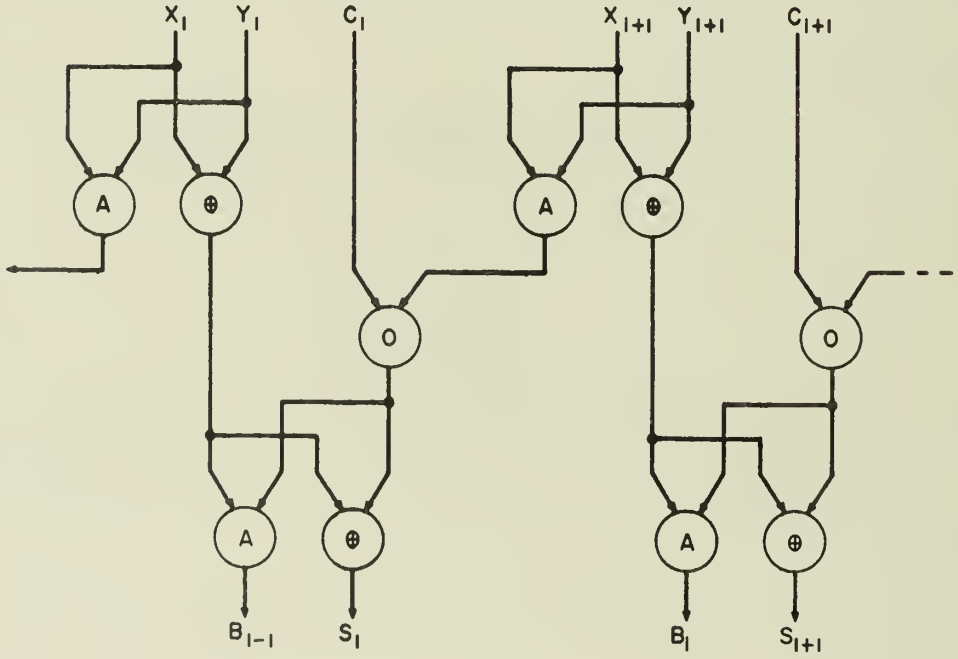


FIGURE 21b. CARRY - SAVE ADDER LOGIC

so both S_i and b_{i-1} cannot both be ones simultaneously. For this reason, point B is chosen as the break point. The function b is the controlling expression for this point in the carry path.

The fact that $S_i \cdot b_{i-1} = 0$ will simplify expressions for the sum digit. Since S_i becomes the X_i for the next addition, this relation will hold if b_{i-1} is stored rather than propagated. The stored b_{i-1} will appear as c_{i-1} during the next addition. Figure 21b shows the logic for two stages of a separate carry-storage adder. Its pseudo-addition time is a fixed four delays (using NANDS).

For carry-assimilation, it is assumed that an adder with the structure of Figure 21a is needed with inputs X_i and C_i, with internal (ripple) carry d_i, and sum digit output S_i. From the relation $S_i \cdot b_{i-1} = 0$ given above, and other relations derived from it, it can be shown [13] that the carry-assimilation equations become

$$S_i = X_i \oplus (c_i \vee d_i)$$

$$d_{i-1} = X_i \cdot (c_i \vee d_i)$$

The adder of Figure 21b can be employed to form the sum digits if the d_i in the above equation are generated first. To use the adder, the Y_i inputs to the AND circuits are set to zero; and the Y_i inputs to the EXCLUSIVE OR circuits are set equivalent to $d_i \cdot \bar{c}_i$. The equation for S_i then beomes

$$S_i = X_i \oplus Y_i \oplus (X_{i+1} \cdot Y_{i+1} \vee C_i)$$

or

$$S_i = X_i \oplus d_i \cdot \bar{C}_i \oplus C_i$$

which can be simplified to

$$S_i = X_i \oplus d_i \oplus c_i = X_i \oplus (d_i \vee c_i)$$

also,

$$d_{i-1} = X_i \cdot (c_i \vee d_i)$$

The carry propagation implied by the last recursion relation may be speeded up by any of the techniques of Sections 3 through 7.

There is no way the add time of the basic carry storage adder can be decreased by rearranging or adding to the logic. It can only operate faster through the use of faster circuit elements. It uses exactly the same number of components as the ripple carry adder, exclusive of carry propagation logic and the extra registers needed. It "adds" in as short a time as is logically possible.

8.3 Summary

The carry-save adder finds its major application in machines with operations in which a large number of additions are performed in series, without an intervening store order which would require the assimilation of carries. Such an operation is multiplication, which may require n additions with an n-bit number. Using a carry-save adder reduces the add time, and, hence, the multiply time to a minimum, since the pseudo-addition amounts to little more than a data-transfer between the X, Y, and C registers and the S and B registers. The adder is, however, rather unhandy to use for a single addition requiring carry-assimilation.

The separate carry-storage adder, when used in the proper manner, will give high-speed addition at minimal cost.

9. SUMMARY OF ADDER-TYPES

The one-position binary full adder was initially introduced. This basic building-block was then employed in the logical construction of an n-bit ripple carry adder, the first type of adder used in electronic computers.

It was seen that the time required for carry propagation in the ripple carry adder is excessive (2n delays for an n-bit adder) compared with the time necessary for sum generation (four delays) in each bit position. Means for reducing this propagation time were then presented.

One of the first ways devised for speeding carries on their way was by using lookahead logic. In this method, each bit position "looks ahead" to lower-ordered positions to see what the carry into it will be, and then forms the sum bit accordingly. By varying the complexity and "reach" of the lookahead logic, in combination with ripple carry logic, the carry lookahead adder can be made as fast as desired, with a limit of eight delays being the maximum speed of operation attainable.

Closely resembling the lookahead adder is the carry skip adder. This method uses signals from each bit position in a group to enable an input carry to "skip" around the group, if conditions are such that the carry would otherwise propagate through. If the carry is not "skipped" around the group, it ripples through until it is absorbed. As the complexity of the skip logic increases, the adder looks more and more like a lookahead adder, until, in the limit, both adder types are almost identical.

The conditional sum adder takes a new approach to addition. This adder cares not what the input carry to a group of bits will be, but generates two sum groups and two output carries from the group.

The proper sum and carry bits are then selected when the true input carry becomes known. By using lookahead techniques within and between the groups, the conditional sum adder's speed can be increased to a maximum of five NAND delays plus the delays of three AND-OR cascades. This adder, one of the fastest available, does not degenerate into another adder type when full carry lookahead is used.

The above mentioned adders are all fixed-time adders, i.e., they complete their additions within a predetermined, invariant period of time. The asynchronous adder, a variable-time device, knows no fixed limits for add time. This adder takes advantage of the fact that the average maximum ripple carry length is less than $\log_2 n$ positions for an n-bit number. The carry propagation path is monitored during addition, and when carry propagation is completed, regardless of time consumed, an "addition completed" signal is generated and sent to the control logic. This adder type uses no carry speed-up logic. It works best, as one might suspect, in an asynchronous, or speed-independent system. In a clocked, or synchronous system, or where a tight bound on add time is required, the use of the asynchronous adder may well be questioned.

The last adder variant mentioned, the carry-save or separate carry-storage adder, is not a true adder since the carries generated in each bit position are not immediately assimilated as in other adder types, but are stored separately from the pseudo-sum. This adder finds use when a long sequence of additions must take place without the necessity for carry propagation. When carry assimilation becomes necessary, any of the above techniques may be used to speed up the process.

9.1 Adder Design Notes

When the designer approaches the problem of selecting and designing an adder for operation in a given system, he must consider many

factors before making the final choice. Some things to take notice of are

a) Number of adder stages needed, i.e., what is the word length?

b) Speed of the system. Must the adder complete its job quickly, or would a ripple carry adder prove adequate?

c) System timing. Is it synchronous (clocked) or asynchronous?

d) Circuit availability and capability. What types of circuits are available for use? What are the circuits' fanin and fanout limitations and their speed of operation?

e) Number representation. Will numbers be represented internal to the system in two's complement, BCD code, floating point, etc.?

f) Register arrangement. How many and what types of registers and gating logic are available for the adder to use?

g) Multiple adders. Two different types of adders may be justified if large numbers of additions as well as many multiplications are anticipated.

9.2 Future

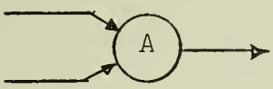
What does the future hold for adders in terms of speed and logical advances? Present adders can be quite complex as one can see by looking at the conditional sum adder. A radix-3 adder has been described [14]. This is another approach to addition, although it is of little use in a binary system. Perhaps future speed increases will be realized only through the development of multi-state and faster-operating devices.

Ten years ago the add time barrier -- the time needed to complete an addition -- was around 100 nsec (100×10^{-9} sec.). Today it is of the order of 10 nsec. This barrier can be penetrated today only through the use of "exotic" devices. An 8-bit ripple carry adder using very high-speed transistors and tunnel diodes has been constructed [1] which can produce the correct sum output in 3 to 5 nsec. With special devices and higher density packaging, the ripple carry delay per stage can be reduced to 0.2 nsec or less.

APPENDIX 1. DEFINITIONS OF LOGIC SYMBOLS

The symbols and notation used in this paper and their truth table definitions are presented below.

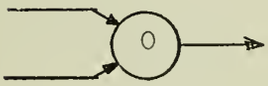
AND:



$Z = A \cdot B$

A	B	Z
0	0	0
0	1	0
1	0	0
1	1	1

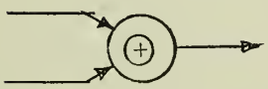
OR:



$Z = A \vee B$

A	B	Z
0	0	0
0	1	1
1	0	1
1	1	1

EXCLUSIVE OR:



$Z = A \oplus B$

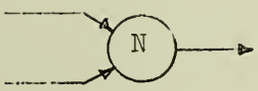
A	B	Z
0	0	0
0	1	1
1	0	1
1	1	0

EQUIVALENCE:

$Z = A \bar{\oplus} B$

A	B	Z
0	0	1
0	1	0
1	0	0
1	1	1

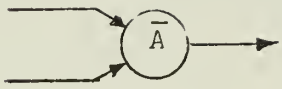
NOT:



$Z = \bar{A}$

A	Z
0	1
1	0

NAND:



$$Z = \overline{A \cdot B}$$

A	B	Z
0	0	1
0	1	1
1	0	1
1	1	0

APPENDIX 2. HARDWARE AND TIMING COMPARISONS

The chart on the following page gives a comparison between modifications of the various adder types and the ripple carry adder. Each adder was designed using only NAND circuits with no limitations on fanin or fanout. Each NAND is assumed to have a time delay of one unit, regardless of its loading. Many NANDs are counted in the chart which do not appear in the logic drawings when a reference to a figure is given.

The number of inputs to the NANDs is an indication of the number of logic diodes needed, while the count of NANDs gives an idea of the number of transistors or tubes necessary to construct the adder.

The chart gives the impression that the carry skip adder yields the greatest speed increase for the smallest additional cost. This is possibly true if NAND circuits alone are used. However, if AND-OR circuits are used, the "cost" of the conditional sum adder (as well as other types) will be greatly reduced with an accompanying <u>increase</u> in speed. The numbers on the chart should be taken only as an indication of the relative costs and speeds of the various adder types since these factors will vary widely depending on the detailed logical structure of the adder and especially on the characteristics of the circuit elements used.

SPEED-HARDWARE COMPARISONS FOR VARIOUS TYPES OF 40-BIT ADDERS

	Adder Type and Variant	Total Number of NANDS[1]	% increase over ripple carry adder	Total Number of Inputs[1]	% increase over ripple carry adder	Max. Number of Delays[1]	Speed increase over ripple carry adder
Ripple Carry	(Fig. 4)	360	--	640	--	80	--
Carry Lookahead	Full lookahead in 4-bit groups; ripple carry between groups (Fig. 5b)	500	39	1000	56	24	3.3
	Full lookahead within (Fig. 5b) and between 4-bit groups (Fig. 12)	565[2]	57	1297	103	8	10
Carry Skip	Skip around 4-bit groups; ripple carry within groups (Fig. 8a)	370	4	700	9	32	2.5
	Skip around variable sized groups; ripple carry within (Fig. 8b)	368	3	696	9	26	3.1
	Full carry skip within 4-bit groups; ripple carry between groups (Fig. 9)	420	17	910	42	24	3.3
	Full carry skip within (Fig. 9) and among 4-bit groups (Fig. 12)	485	35	1200	88	8	10
Conditional Sum	Ripple carry internal to and between 4-bit groups (Fig. 14)	628	74	1290	101	22	3.6
	Ripple carry internal to 4-bit groups; full lookahead between groups (Fig. 16)	640	75	1590	148	9	8.9
	Full carry lookahead within and between 4-bit groups (Figs. 12 and 16)	780	88	1690	164	8	10

(Footnotes on next page)

Adder Type and Variant	Total Number of NANDS[1]	% increase over ripple carry adder	Total Number of Inputs[1]	% increase over ripple carry adder	Max. Number of Delays[1]	Speed increase over ripple carry adder
Asynchronous	481	34	920	44	4 min. 82 max.	20 0.98

1. Figures given are worst-case. An actual design would doubtless use fewer NANDs than the chart indicates, and fewer inputs, and achieve a higher speed.

2. In obtaining these numbers, it was assumed fanins of 15 and fanouts of 10 or more were acceptable.

BIBLIOGRAPHY

[1] Amodei, J., "High-Speed Adders and Comparators Using Transistors and Tunnel Diodes," IEEE Transactions on Electronic Computers, EC-6, October 1964, pp. 563-575.

[2] Beckman, A., "Developments in the Logical Organization of Computer Arithmetic and Control Units," Proceedings of the IRE, 49, January 1961, pp. 60-66.

[3] Bedrij, O. J., "Carry Select Adder," IRE Transactions on Electronic Computers, EC-11, June 1962, pp. 340-346.

[4] Burks, A. W., H. Goldstine, and J. Von Neumann, " Preliminary Discussion of the Logical Design of an Electronic Computing Instrument," The Institute for Advanced Study, Princeton, New Jersey, 1947.

[5] Flores, Ivan, The Logic of Computer Arithmetic, Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1963, pp. 70-119.

[6] Gilchrist, B., J. H. Pomerene, and S. Y. Wong, "Fast Carry Logic for Digital Computers," IRE Transactions on Electronic Computers, EC-4, December 1955, pp. 133-136.

[7] Hendrickson, H. C., "Fast High-Accuracy Binary Parallel Addition," IRE Transactions on Electronic Computers, EC-9, December 1960, pp. 465-466.

[8] Hsiao, M., "The Carry-Dependent Sum Adder," IEEE Transactions on Electronic Computers, EC-12, June 1963, pp. 265-268.

[9] Lehman, M. and N. Burla, "Skip Techniques for High-Speed Carry-Propagation in Binary Arithmetic Units," IRE Transactions on Electronic Computers, EC-10, December 1961, pp. 691-698.

[10] MacSorley, O. L., "High Speed Arithmetic in Binary Computers," Proceedings of the IRE, 49, January 1961, pp. 67-71.

[11] Metze, G., "A Binary, Parallel, Asynchronous Arithmetic Unit with Separate Carry or Borrow Storage," Department of Computer Science, University of Illinois, Urbana, Illinois 61803, File No. 224, July 1957. (Presented at the 12th National Meeting of the ACM, Houston, Texas.)

[12] Morgan, C. and D. Jarvis, "Transistor Logic Using Current Switching and Routing Techniques and Its Application to a Fast-Carry Propagation Adder," Proceedings of the IEE, Pt. B, 106, September 1959, pp. 467-468.

[13] Robertson, J. E., "Theory of Computer Arithmetic Employed in the Design of the New Computer at the University of Illinois," Department of Computer Science, University of Illinois, Urbana, Illinois 61803, File No. 319, June 1960.

[14] Rozmarich, T. A., "Synthesis of Three-Level Logic Circuits with Application to a Radix Three Computer Arithmetic Unit," Department of Computer Science, University of Illinois, Urbana, Illinois 61803, Report No. 181, June 1965.

[15] Salter, Forrest, "High-Speed Transistorized Adder for a Digital Computer," IRE Transactions on Electronic Computers, EC-9, December 1960, p. 461.

[16] Sklansky, J., "Conditional Sum Addition Logic," IRE Transactions on Electronic Computers, EC-9, June 1960, pp. 226-231.

[17] _____, "Ultimate - Speed Adders," IEEE Transactions on Electronic Computers, EC-12, April 1963, pp. 142-148.

[18] Weinberger, A. and J. Smith, "A One Microsecond Adder Using One Megacycle Circuitry," IRE Transactions on Electronic Computers, EC-5, June 1956, pp. 65-73.

UNIVERSITY OF ILLINOIS-URBANA
3 0112 103707086